The University of York Department of Computer Science

Submitted in part fulfilment for the degree of BEng.

3D Reconstruction of Scenes from multiple Photos

Tom SF Haines

17th March 2005

Supervisor: Dr. Richard Wilson

Number of words = 17322, as counted by `wc -w`.
This includes the body of the report but none of the appendices.

Abstract

3D reconstruction is the creation of recognisable virtual 3D scenes directly from reality, without the need for an artist. It is a commercial reality, for example [1, 2, 3], however the services these and other companies provide are not cheap[1] due to the specialist hardware requirement and the limited market that follows from the price. Research now focuses on 3D reconstruction with cheap commodity hardware under difficult conditions, as opposed to complex sensors and controlled conditions. Systems capable of working within such constraints have emerged[5]. This project attempts to produce such a system.

[1] Direct Dimensions[1] products are based on specialist hardware, which starts at $10000 and reaches $150000, before software and training is taken into account[4].

Contents

List of Figures

1 Introduction

This section provides an overview of the project, followed by the motivations behind the project and then the structure of this document as a whole.

1.1 Overview

3D reconstruction from various types of input is a long standing problem in computer vision. Under controlled circumstances the problem is manageable though by no means perfect, and as such current research is focused on removing the constraints of current solutions. A few common constraints as they apply to particular techniques follow:

- **Known background**. One of the most reliable classes of solution, space carving, distinguishes between the foreground and the background, using many such classified views to carve the shape of the viewed object. This requires that the background be identifiable, such as a blue screen or a background plate taken before the object was introduced.

- **Transparency and Reflections**. Efforts have gone into managing scenes with these properties[6], however most current systems fail on encountering either, so its best to avoid such scenes. This unfortunately covers the majority of real world objects.

- **Known sensors**. To create a proper Euclidean reconstruction using input from cameras you need to know the properties of the cameras, otherwise the returned geometry will be distorted. This used to involve long and complex processes, but can now be done with little effort[7], and can even be done from the scene being captured[8].

- **Controlled light**. Some systems work out the shape of a scene based on light, so called *shape from shading*, for this to work details about the light sources usually have to be known. This is often done by controlling the lighting, in a studio for example.

- **Scene constraints**. If properties about the scene in question are known the work required to model them can be considerably reduced. For instance, if the scene is made up of only cuboid structures then only flat surfaces at right angles to each other need be considered.

The most advanced 3D reconstruction *device* we know of is are own eyes[1], so the ultimate goal is to match and then surpass this[2]. Such technology is still far away. It is possible however to work within the many constraints and limitations of current technology to create a system that can be casually[3] used to model a wide variety of scenes.

The biggest issue facing current systems is reliability/robustness. Most algorithms have countless failure scenarios and when they do work the output often both transfers the noise in the input and adds mistakes of its own, so each part of the system has to cope with large numbers of errors, sometimes over half of the data. Many of these errors survive to the final output (Even if detected and deleted, leaving holes.) making the data useless for most practical purposes.

1.2 Motivation

In the media rich world we live in today 3D models sourced from reality have countless uses, to name a few

- Military training simulations[4].
- Interfaces[5].
- TV/Film Special Effects[6].
- Computer Games[7].

In all of the above fields 3D models are created by artists and 3D scanners when they can afford it. The ability to reliably create 3D models using cheap digital cameras would be invaluable, especially in the amateur equivalents of these fields. It is unrealistic to expect to produce such a system with current technology within the time constraints of this project, but the aim is to get as close as possible.

[1] Or to be precise, our eyes in combination with our brain and our ability to actively direct our eyes with the muscles in our neck.

[2] It is prudent to note that whilst we as humans understand the scene before us in a 3D sense, we do not have a 3D model such as we are aiming to achieve, without specifically thinking about it at any rate. We, as humans, also understand the scene in front of us at a high level, whilst a 3D model is only an understanding of shape and colour.

[3] By casually I mean without planning, such as booking a blue screen or setting up specialist lighting. It is not intended to imply the task being easy or reliable.

[4] http://www.breakawayfederal.com/

[5] http://javadesktop.org/articles/LookingGlass/index.html

[6] http://ilm.com/,http://www.pixar.com/

[7] http://www.half-life2.com/,http://doom3.com/

1.3 Structure

This document is divided up as follows:

- **Chapter 2** *Concepts* covers the physical, geometrical and mathematical principles the project is built on. This includes various formula required by the system that do not warrant detailed discussion.
- **Chapter 3** *Proccessing Models* discusses the types of algorithm and how they can be combined to produce a working system.
- **Chapter 4** *Design* covers the design of the implemented system.
- **Chapter 5** *System Evaluation* covers standard testing of the code and finding the limits of the system i.e. where it fails.
- **Chapter 6** *Conclusion* briefly summarises the system and discusses both its failings and potential further work.
- **Appendix A** *Aegle Structure* covers the system implementation in detail.
- **Appendix B** *Aegle Operations* covers the implemented algorithms and how to use them.
- **Appendix C** *Testing* contains the scripts and results of testing that are not included in the main body.

2 Concepts

This chapter is a review of the many concepts on which this project is based. It covers many disparate subjects and by nature lacks the structure of latter chapters.

2.1 2D & 3D Representation

This section discusses ways of representing 2D and 3D data, being as they are the respective input and output data types of this project.

2D images on a computer have two representations:

- **Pixel based**. Pixels (Picture Elements) are individual samples of the 'value' of a particular position, typically on a square grid. In a typical colour digital image this will be the three components; Red, Green and Blue [RGB]. Pixel based storage has to be used for sampled data, such as that captured by a digital camera.
- **Vector based**. Vector based graphics are stored as a sequence of mathematically constructed primitives. For instance, a line segment can be stored as a pair of position vectors indicating endpoints, whilst a circle can be stored as a position vector for its centre and a radius.

in turn, both representations extend to 3D:

- **Voxel based**. The extension from 2D pixels to 3D voxels (Volume Elements) simply involves redefining the sampling pattern to cover three dimensions instead of two. It is often useful to visualise such an arrangement as 2D images stacked in the Z dimension.
- **Vector based**. The coordinate system has to work in three dimensions; extra primitives are usually provided to represent solids. For instance, a sphere could be represented as a position vector and radius. The lowest common denominator of 3D representation, which is almost invariably supported, is a triangle mesh. This is a set of triangles, each represented by three position vectors for the corners.

Ultimately 3D output from the system must be in a tightly constrained vector format due to the limitations of 3D rendering hardware. (That is presuming a custom renderer is not written. Time constraints forbid this here.) Algorithms

can produce data in many formats, including less constrained geometric primitives from those which can be rendered, in such cases the data needs to be converted to the required format for rendering. For instance the Marching Cubes[9] algorithm would need to be applied if voxels were the output. Once the data is in the right format problems can be encountered if too much data is present for the hardware to cope with, so data reduction becomes an issue. Generally vector based representations are preferred for 3D data, for many reasons that ultimately come down to the ability to view 3D data from many positions, so considerably more information is needed to maintain correctness in all views. The memory consumption of any reasonable sized voxel field soon becomes prohibitive, and unlike images where a single colour value is enough 3D data requires material information so it can change according to the viewing parameters.

2.2 Image Processing

This section covers a select set of image processing techniques that are used by this project.

2.2.1 Corners

An edge is where a transition occurs in an image. Edge detectors attempt to find such transitions; these usually occur at the boundaries of objects, shadows and regions of texture. A common approach is to find the zero crossings of the second derivative of the image. A corner can be defined as where two edges meet/cross.

There are many corner detectors, a popular one is the Harris corner detector[10], which calculates a corner response function for each pixel as

$$R = \det \boldsymbol{M} - k\text{trace}(\boldsymbol{M})^2 \tag{2.1}$$

where $k = 0.04$ and

$$\boldsymbol{M} = \sum_{W} \begin{bmatrix} \frac{\delta I}{\delta x} \\ \frac{\delta I}{\delta y} \end{bmatrix} \begin{bmatrix} \frac{\delta I}{\delta x} & \frac{\delta I}{\delta y} \end{bmatrix} w(x,y) \tag{2.2}$$

The sum is over a window around the pixel in question, where $w(x,y)$ is a weighting function over the window, suggested as a Gaussian kernel with $sd = 0.7$ by [5]. A corner is found at every local maximum of this function, this produces too many to work with so they have to be pruned. Pruning can be done by selecting the corners with the largest response.

2.2.2 Normalised Cross Correlation

Normalised Cross Correlation[11] [NCC] returns the *similarity* of two windows of pixels. For our purposes high similarity between two windows centred on

corners gives an indication that the corners match. It is given as

$$S = \frac{\sum_W (I(x,y) - \bar{I})(J(x,y) - \bar{J})w(x,y)}{\sqrt{\sum_W \left[(I(x,y) - \bar{I})w(x,y)\right]^2 \sum_W \left[(J(x,y) - \bar{J})w(x,y)\right]^2}} \tag{2.3}$$

where $\bar{I} = \frac{1}{w}\sum_W I(x,y)$, $\bar{J} = \frac{1}{w}\sum_W J(x,y)$, $I(x,y)$ is the value of a pixel from window 1 and $J(x,y)$ is the value of a pixel from window 2. $w(x,y)$ is a weighting function.

2.3 Geometry

The first part of this section covers the use of vectors and matrices with respect to Euclidean geometry as a motivation for homogeneous coordinates, the following part. Projective geometry is covered in the third part, as the basis for most of the systems formula.

2.3.1 Vectors & Matrices

A 2-dimensional vector, $\boldsymbol{X} = \begin{bmatrix} x \\ y \end{bmatrix}$ is used to represent a particular location, with respect to $O = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, the origin. As a matter of notational convenience this is also written as $\begin{bmatrix} x & y \end{bmatrix}^T$. For 3D the obvious extension applies, $\begin{bmatrix} x & y & z \end{bmatrix}^T$.

A matrix can represent a transformation, $\boldsymbol{X}' = \boldsymbol{T}\boldsymbol{X}$ where $\boldsymbol{X}'$ is the transformed point, $\boldsymbol{X}$.

$$\boldsymbol{X}' = \begin{bmatrix} s & 0 \\ 0 & s \end{bmatrix} \boldsymbol{X} \tag{2.4}$$

$$\boldsymbol{X}' = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix} \boldsymbol{X} \tag{2.5}$$

Equation 2.4 is a scaling with respect to the origin, where s is a scaling factor; Equation 2.5 is a rotation around the origin (Counter-clockwise) of θ radians. There are other transformations, such as shears not mentioned here. The same transformations naturally extend to 3D. A sequence of these transformations can be composed together into a single matrix by multiplication, however there is one transformation missing, translation. Equation 2.6 achieves a translation of $\begin{bmatrix} u & v \end{bmatrix}^T$, but it does this using addition, and cannot be composed with other transformations.

$$\boldsymbol{X}' = \boldsymbol{X} + \begin{bmatrix} u \\ v \end{bmatrix} \tag{2.6}$$

2.3.2 Homogeneous Coordinates

Homogeneous coordinates are a technique that, among other things, can be used for expressing translations with matrices. The vectors used to represent a position are extended with an extra component, w, so for 2D $[\,x \quad y \quad w\,]^T$ and for 3D $[\,x \quad y \quad z \quad w\,]^T$. (For Euclidean geometry $w \neq 0$. In the next section this constraint is relaxed.) To transfer to and from homogeneous coordinates use the relation $[\,x \quad y \quad w\,]^T = [\,x/w \quad y/w\,]^T$.

2D transformations are now represented by 3x3 matrices, with Equation 2.7 being a scaling and Equation 2.8 a rotation.

$$\boldsymbol{X}' = \begin{bmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & 1 \end{bmatrix} \boldsymbol{X} \tag{2.7}$$

$$\boldsymbol{X}' = \begin{bmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \boldsymbol{X} \tag{2.8}$$

$$\boldsymbol{X}' = \begin{bmatrix} 1 & 0 & u \\ 0 & 1 & v \\ 0 & 0 & 1 \end{bmatrix} \boldsymbol{X} \tag{2.9}$$

The advantage is that now we can represent a translation as a matrix, as in Equation 2.9. This means we can now compose all these types of relation into a single matrix. Note that for Euclidean transformations of 2D the last row will always be $[\,0 \quad 0 \quad 1\,]$, or equivalently $[\,0 \quad 0 \quad 0 \quad 1\,]$ for 3D.

2.3.3 Projective Geometry

This section is based on [12, 5], due to the complexity of projective geometry this is a light tour at best, for more details see the references.

Projective geometry is fundamentally different from Euclidean geometry, having one different axiom. They share the first four axioms[1], but the fifth is different. For Euclidean geometry the 5^{th} axiom implies that there can be parallel lines that never intersect, for projective geometry it states that all lines intersect. The consequences of this are many, but the important consequence of using projective geometry is in regards to the transformations. Euclidean geometry allows only rotation and translation transforms whilst projective additionally allows scaling, shear and perspective transformations[2]. This affects what can be considered invariant under transformation in these geometries, under projective geometry very little survives an arbitrary transformation, in fact only incidence (If two objects occupy the same space before the transform they will do so after it.) and the cross-ratio invariant[12]. Under Euclidean geometry almost everything survives - distances, angles, parallelism.

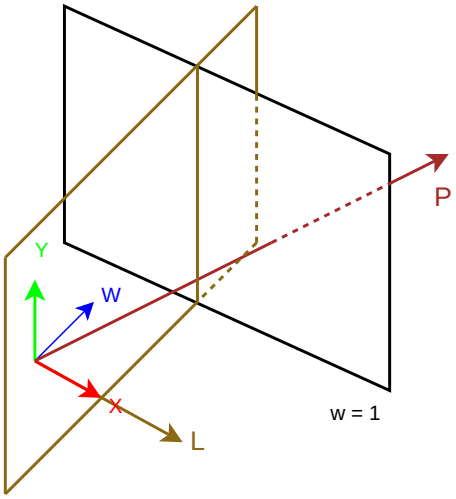


Figure 2.1: Projective Geometry Point & Line

[1] Taken directly from http://en.wikipedia.org/wiki/Euclidean_geometry:

1. Any two points can be joined by a straight line.
2. Any straight line segment can be extended indefinitely in a straight line.
3. Given any straight line segment, a circle can be drawn having the segment as radius and one endpoint as centre.
4. All right angles are congruent.

[2] The are geometries between Euclidean and projective. Similarity allows rotation, translation and scaling, affine allows everything that similarity does plus shearing.

For practical purposes projective geometry is represented using homogeneous coordinates. For 2D you can visualise the homogeneous coordinates as being vectors in 3D space with w as the extra dimension. Where a point vector intercepts the plane $w = 1$ is the point in Euclidean space. Figure 2.1 demonstrates this with line P. A consequence of this is that equality is with a scale factor, i.e. $\forall \alpha, \alpha \neq 0 \Rightarrow [\, x \quad y \quad w \,]^T = [\, \alpha x \quad \alpha y \quad \alpha w \,]^T$. A line is represented by a plane, its position in Euclidean space being where it intercepts the $w = 1$ plane. The plane is represented by a vector perpendicular to the plane, so it has the same representation as a point, $[\, x \quad y \quad w \,]^T$, as indicated by L in Figure 2.1. Equality with a scale factor also applies to lines. Transformations are again represented by matrices as in subsection 2.3.2, however the last row can now be any arbitrary value for projective transforms. Transformation matrices are also equal with a scale factor.

Projective Geometry limits what can be calculated, as things like angles and distance no longer mean anything. Given points as $\boldsymbol{X}_n = [\, x \quad y \quad w \,]^T$ then the line $\boldsymbol{L}_n = [\, x \quad y \quad w \,]^T$ that passes through two of them is $\boldsymbol{L} = [\boldsymbol{X}_1]_x \boldsymbol{X}_2$ where $[\boldsymbol{X}_1]_x$ indicates the cross product[3]. The reverse also holds, given two lines then $\boldsymbol{X} = [\boldsymbol{L}_1]_x \boldsymbol{L}_2$ where $\boldsymbol{X}$ is the point of intersection for the two lines. A line and point intersect if $\boldsymbol{P}^T \boldsymbol{L} = 0$. Points at infinity are represented by vectors with $w = 0$, there is also a line at infinity, $\boldsymbol{L} = [\, 0, 0, \alpha \,]^T, \alpha \neq 0$.

2.4 Sensors

To build a model of reality within a computer one first has to obtain information about that reality. A non-exhaustive list of relevant sensors follows:

- **Human input**. As obvious as it is to state, a human can act as a sensor and input data into a computer. Human beings have an extremely good sense of the 3D world, but not a very good sense of how to transfer this information into a computer. In addition, such work is time consuming, tedious and expensive to pay for[4].

  One can view the activity of getting a computer to automatically produce a 3D reconstruction as one of automation, of removing the human being from the loop. Total removal is not yet possible, if only because a human has to operate the sensors used instead, but some tasks still lend themselves to human involvement. Examples of such tasks are providing one

[3] Given a vector $\boldsymbol{V} = [\, v_1 \quad v_2 \quad v_3 \,]^T$ then $[\boldsymbol{V}]_x = \begin{bmatrix} 0 & -v_3 & v_2 \\ v_3 & 0 & -v_1 \\ -v_2 & v_1 & 0 \end{bmatrix}$. If written as $[\boldsymbol{A}]_x \boldsymbol{B}$ for two vectors $\boldsymbol{A}$ and $\boldsymbol{B}$ then this is the cross product of the two vectors.

[4] In most subject areas pointing out that humans make mistakes is traditionally done here. However, from the results obtained they are models of perfection compared to what can currently be achieved in computer vision.

shot values, such as the real world distance between two points to scale a model; or providing approximate initial data for refinement by the computer, such as the spatial relationship between two sensor readings. The key area where human involvement is vital however is where there are currently no algorithms or the algorithms that exist are of limited capability. An example of this is in deciding which sensor readings are related; that two images of the same object are of the same object, and that two images of different objects are not.

- **Cameras**. Cameras project the 3D world onto a 2D plane and record this projection, thereby losing any depth information. The vast majority of cameras available capture what we as humans perceive. Cameras can be constructed to capture most forms of light radiation however, for instance to capture x-rays in a hospital. Video cameras are a form of camera that captures many frames at regular intervals, to create the illusion of movement when played back. Technical details on cameras are given in the next section, 2.5.

- **3D Scanners**. For the purpose of this discussion the term *3D Scanner* means anything that produces 3D data directly, not just laser scanners. Laser scanners are the common tool however. They work by projecting a laser beam onto the object and recording the shape the beam makes on the object using a sensor (Often a camera) in a known relation with the laser beam source, as indicated in Figure 2.2. You then extract the line by taking two readings, one with the laser and one without, then subtracting the two to leave only the line. An analysis of the line with the knowledge of the relative geometry between the laser and the sensor will result in 3D data. If the sensor is a camera then colour data can be extracted simultaneously, however it is often the case that such data is not retrieved so colour has to be extracted and then aligned with the 3D data using other techniques. It is also often the case that complete object coverage is achieved with multiple scans, so these separate models have to be latter aligned. As previously mentioned, such tools can be very expensive[4], though cheap once can be constructed if you have the time and quality is not an issue[13].

  To give a second example a Coordinate Measurement Machine[14] [CMM] is a human operated device that measures the position of a 'pen'. The operator can record the shape of a surface by simply moving the pen across it whilst recording. The methods of determining the position of the pen at any given moment vary, but examples would be having the pen on the end of an arm that senses the angle of its joints, or connecting the pen to distance sensors from 3 points to triangulate position. This could be as simple as three lengths of string being kept taught using springs, with newton meters to measure how far the string is extended.

  There are other examples, such as Radar[15] and Magnetic Resonance Imag-

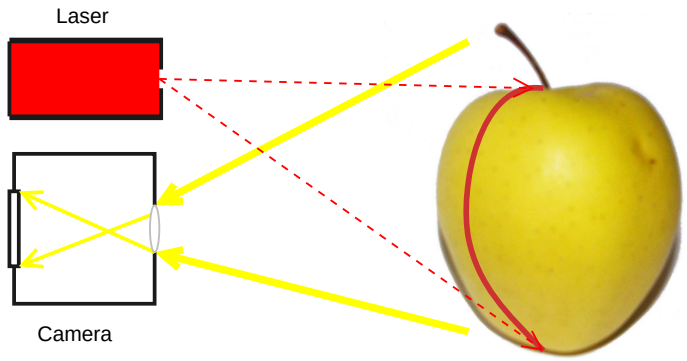


Figure 2.2: A diagramatic representation of a laser scanner. At the top you have a laser that outputs a line onto the object, at the bottom a camera to record the position of the laser on the object. To scan the entire object either the laser has to move or the object has to move.

ing [MRI]. It is key to note that any given implementation of 3D scanning technology has a targeted scale and resolution. This means that you use a different device to scan a person as you do a car, this lack of general applicability alongside the expense limits this technology to a set of niche markets.

2.5 The Camera

A camera (Or an eyeball.) works by focusing light captured from the scene in front of it onto a sensor at the back of the device, as shown in Figure 2.3. A lens

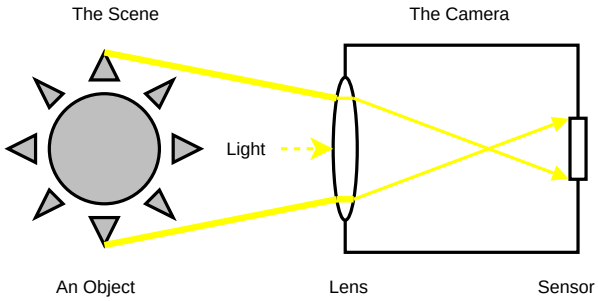


Figure 2.3: A camera with a lens

is a complex entity, and the set of lenses found in a real camera are an extremely complex entity, so for the purpose of modelling the behaviour of a camera it is easier to use a pinhole camera with an infinitely small aperture. This approxi-

mation is generally close enough, though for high precision modelling the various distortions that can occur need to be taken into account. (See section 2.6) Figure 2.4 illustrates a Pinhole camera with a single ray of light marked on. By

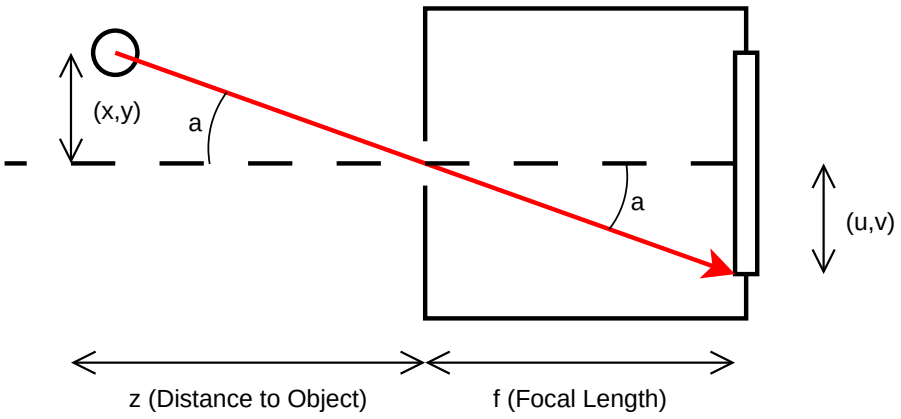


Figure 2.4: A simplified model of a camera

similar triangles

$$u = \frac{xf}{z} \qquad \& \qquad v = \frac{yf}{z} \tag{2.10}$$

These equations can be folded into a matrix using homogeneous coordinates so additional camera properties may be added

$$\boldsymbol{s} = \boldsymbol{PX} \tag{2.11}$$

where $\boldsymbol{X} = [\,x \quad y \quad z \quad 1\,]^T$, $\boldsymbol{s} = [\,u \quad v \quad 1\,]^T$ and $\boldsymbol{P}$ is the $3x4$ camera projection matrix.

The projection matrix can be decomposed into two parts, intrinsic and extrinsic. The intrinsic properties of a given camera don't change[5] and represent camera specific properties such as the field of view. The extrinsic parameters are the position and orientation of a camera relative to some set position and orientation. This origin is usually set to be one of the cameras with all others relative to this base measurement, or to the centre of a targeting object visible in all views. This decomposition can be further subdivided as in Equation 2.12 (2.12 is based on [16, 5]).

$$\boldsymbol{P} = \underbrace{\begin{bmatrix} \frac{f}{P_x} & \tan(\alpha)\frac{f}{P_y} & c_x \\ 0 & \frac{f}{P_y} & c_y \\ 0 & 0 & 1 \end{bmatrix}}_{\textit{intrinsic matrix}} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \underbrace{\left[\begin{array}{c|c} \boldsymbol{R}^T & -R^T\boldsymbol{T} \\ \hline 0_3^T & 1 \end{array}\right]}_{\textit{extrinsic matrix}} \tag{2.12}$$

[5] Unfortunately they do change, depending on such things as F-stop (Size of the aperture through which light passes to reach the sensor.), focal length (Zoom) and focusing. Modern cameras change these settings without the users intervention making calibrating for them difficult.

The intrinsic matrix is constructed from several parameters, f is the focal length of the camera (i.e. 28mm, 35mm.), P_x and P_y are the dimensions of the pixels[6], α is the angular skew whilst c_x and c_y form the principal point. The principal point is the centre of the sensor, in theory this is aligned with the centre of the aperture, so if the image is parametrized as [-1,1]x[-1,1] should be $\begin{bmatrix} 0 & 0 \end{bmatrix}^T$. Similarly, the pixels should be perfectly square therefore the skew should be 0. You can calculate a cameras intrinsic matrix from the manufacturers provided specifications, but this does not produce a useful result as workmanship is not perfect and most of the parameters used only make true sense for a pin hole camera, so the manufacturer provides approximate parameters anyway.

The extrinsic matrix is constructed from a rotation ($\boldsymbol{R}$) and a translation ($\boldsymbol{T}$) from the origin to the camera. The projection matrix must contain the inverse, which can be constructed as in Equation 2.12. Given pairs of world coordinates and projected coordinates it is possible to calculate the projection matrix of a camera using SVD.[17, p. 455][8] If the projection matrix is represented as

$$\boldsymbol{P} = \begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \end{bmatrix} \tag{2.13}$$

then for any given match, $\boldsymbol{X} = \begin{bmatrix} x & y & z & 1 \end{bmatrix}^T$, $\boldsymbol{s} = \begin{bmatrix} u & v & 1 \end{bmatrix}^T$

$$\begin{bmatrix} x & y & z & 1 & 0 & 0 & 0 & 0 & -ux & -uy & -uz & -u \\ 0 & 0 & 0 & 0 & x & y & z & 1 & -vx & -vy & -vz & -v \end{bmatrix} \begin{bmatrix} p_{11} \\ p_{12} \\ \vdots \\ p_{33} \\ p_{34} \end{bmatrix} = 0 \tag{2.14}$$

As $\boldsymbol{P}$ has 11 degrees of freedom (4$\boldsymbol{x}$3 homogeneous matrix minus one for equality with a scaling factor.) 6 matches can be used to find the projection matrix

[6] This is the size of a pixel in real-world units. Approximately calculated by taking the size of the CCD[7] and dividing by the number of pixels in each dimension. (This is wrong because not all of the CCDs area is used.)

[7] A digital camera uses a Charged Coupling Device [CCD] to detect light, its equivalent to the film in an analog camera.

[8] Singular Value Decomposition[18] [SVD] decomposes a mxn matrix $\boldsymbol{A}$ such that $\boldsymbol{A} = \boldsymbol{UDV}^T$ where $\boldsymbol{U}$ is a mxn ($m > n$) matrix, $\boldsymbol{V}$ is a nxn matrix and $\boldsymbol{D}$ is a nxn diagonal matrix. Both $\boldsymbol{U}$ and $\boldsymbol{V}$ have orthogonal columns such that $\boldsymbol{UU}^T = \boldsymbol{VV}^T = \boldsymbol{I}$. It works for any given matrix, including non-invertible once.

using SVD[9]

Given the projection matrices $\boldsymbol{P}_n = \begin{bmatrix} \boldsymbol{P}_1^T \\ \boldsymbol{P}_2^T \\ \boldsymbol{P}_3^T \end{bmatrix}$ of two or more cameras and $\boldsymbol{s}_n = [\, u \quad v \quad 1 \,]^T$ where a single point $\boldsymbol{X} = [\, x \quad y \quad z \quad w \,]^T$ projects onto each of them you can obtain $\boldsymbol{X}$ using[19]

$$\begin{bmatrix} u\boldsymbol{P}_3^T - \boldsymbol{P}_1^T \\ v\boldsymbol{P}_3^T - \boldsymbol{P}_2^T \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = 0 \tag{2.15}$$

for each camera and solving with SVD. $\boldsymbol{X}$ has four degrees of freedom so at least two views are required.

2.6 Imaging Errors

There are two types of error caused by the imaging process worth considering; noise from the sensor and distortion from the lens(es).

Noise comes from many sources[20], for practical purposes however it is invariably modelled as a Gaussian distribution around the true value of a given sensor reading.

Lenses cause several distortions but only radial (barrel) distortion, as illustrated by Figure 2.5, is on a large enough scale to be worth removing[7]. Using normalised image coordinates throughout if (x, y) represents undistorted coordinates and (u, v) represents the distorted coordinates then

$$u = x + x[k_1(x^2 + y^2) + k_2(x^2 + y^2)^2] \tag{2.16}$$

$$v = y + y[k_1(x^2 + y^2) + k_2(x^2 + y^2)^2] \tag{2.17}$$

where k_1 and k_2 are the distortion parameters.

[9] Given a set of equations in the form $\boldsymbol{G}_n^T\boldsymbol{H} = 0$ where $\boldsymbol{G}_n$ and $\boldsymbol{H}$ are both vectors then to solve for $\boldsymbol{H}$ you construct the matrix $\boldsymbol{A}$ by stacking $\boldsymbol{G}_n$ i.e. $\boldsymbol{A} = \begin{bmatrix} \boldsymbol{G}_1 \\ \boldsymbol{G}_2 \\ \vdots \\ \boldsymbol{G}_N \end{bmatrix}$. Using SVD such that $\boldsymbol{A} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{V}^T$ then $\boldsymbol{H}$ is the column of $\boldsymbol{V}$ associated with the lowest value of $\boldsymbol{D}$. Note that using this technique you can over specify the number of equations and it will algebraically minimise the error. If over-specifying it is prudent to make sure that the numbers used are all in similar scales otherwise the error minimised will be biased towards the largest scale.

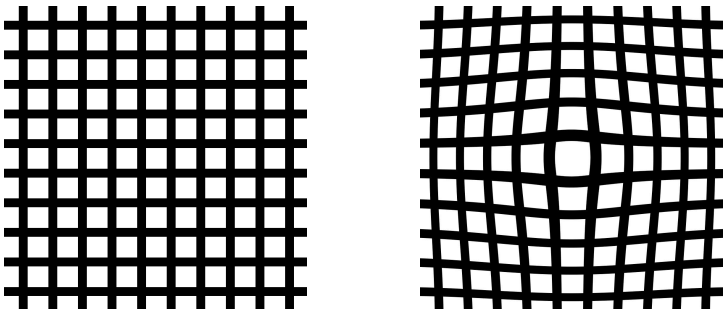


Figure 2.5: The image on the left illustrates a grid with no distortion whilst the image on the right illustrates exaggerated barrel distortion.

2.7 Epipolar Geometry

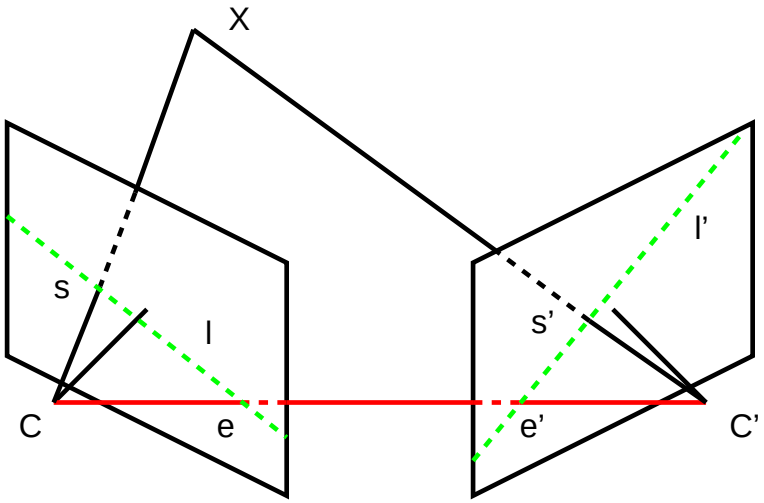


Figure 2.6: The Epipolar arrangment

Working with Figure 2.6, a point $\boldsymbol{X}$ is projected onto two cameras as image points s and s', with the cameras centres of projection as $\boldsymbol{C}$ and $\boldsymbol{C}'$ respectively. The line that passes through s and $\boldsymbol{C}$ and the line that passes through s' and $\boldsymbol{C}'$ intercept at $\boldsymbol{X}$. A plane can be defined to contain these two lines. A third line from $\boldsymbol{C}$ to $\boldsymbol{C}'$ also lies on this plane, with two points where it intercepts the image planes, e and e', referred to as the epipolar points. (e is the projection of point $\boldsymbol{C}'$ onto the camera $\boldsymbol{C}$ and visa-versa.) Where this plane intercepts the image planes you get the lines l and l' (The green dotted lines l and l' from Figure 2.6) which contain the epipolar point and the projection of $\boldsymbol{X}$ onto there respective cameras. The crux of epipolar geometry[16] is that e and e' are constant for any given camera configuration, so if you know the camera configuration and are

given s you know that s' is restricted to a line, l', and visa-versa. This is useful if attempting to match points in two images as we can use epipolar geometry to reduce the search space from 2D to 1D.

A 3x3 fundamental matrix $\boldsymbol{F}$ can be defined to expresses this constraint, such that

$$s'^T \boldsymbol{F} s = 0 \tag{2.18}$$

$$l' = \boldsymbol{F} s \tag{2.19}$$

$$l = \boldsymbol{F}^T s' \tag{2.20}$$

In addition, the epipole e' is the left null space and e is the right null space[10] of $\boldsymbol{F}$.

If we know the projection matrices for two cameras $\boldsymbol{P}$ and $\boldsymbol{P}'$ we can calculate the fundamental matrix using

$$\boldsymbol{F} = \left[\boldsymbol{P}'\boldsymbol{C}\right]_x \boldsymbol{P}'\boldsymbol{P}^\dagger \tag{2.21}$$

where $\boldsymbol{P}^\dagger$ is the moore-penrose pseudo inverse[11]. Of more value however is that given a set of matches between points in one image and points in another we can also calculate the fundamental matrix[5]. If the fundamental matrix is defined as

$$\boldsymbol{F} = \begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{bmatrix} \tag{2.22}$$

then for each match, $s = \begin{bmatrix} x & y & 1 \end{bmatrix}$ and $s' = \begin{bmatrix} x' & y' & 1 \end{bmatrix}$ between the two images we have

$$\begin{bmatrix} xx' & yx' & x' & xy' & yy' & y' & x & y & 1 \end{bmatrix} \begin{bmatrix} f_{11} \\ f_{12} \\ \vdots \\ f_{33} \end{bmatrix} = 0 \tag{2.23}$$

The fundamental matrix has 7 degrees of freedom as its rank 2 and invariant to scale[16]. As the rank 2 constraint can not be expressed in the above form it is easier to give it 8+ matches, solve with SVD and then force the rank 2 constraint[12].

From the fundamental matrix it is possible to recover the projection matrices of two cameras *within a projective transformation*[16].

$$\boldsymbol{P} = \left[\begin{array}{c|c} \boldsymbol{I}_3 & \boldsymbol{0} \end{array}\right] \tag{2.24}$$

[10] The left and right null space of a matrix can be found by applying SVD where $\boldsymbol{A} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{V}^T$, the left null space is the column of U associated with the lowest value of D whilst the right null space is the column in V.

[11] The moore-penrose pseudo inverse can be implemented using SVD, $\boldsymbol{A}^\dagger = \boldsymbol{V}\boldsymbol{D}^{-1}\boldsymbol{U}^T$ where $\boldsymbol{A} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{V}^T$ and $\boldsymbol{D}^{-1}$ is calculated by inverting all non-zero elements.

[12] This can be done with SVD, decompose the matrix as $\boldsymbol{F} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{V}^T$, set the third diagonal element of $\boldsymbol{D}$ to 0 then multiply it back to get $\boldsymbol{F}$.

$$\boldsymbol{P}' = \left[\; [e']_x \boldsymbol{F} \;\middle|\; e' \; \right] \tag{2.25}$$

Transforming the resulting matrices to the correct transforms then requires further information. Alternatively, if you have the intrinsic parameters for the cameras you can calculate the projective transformations directly via the essential matrix.

The essential matrix is identical to the fundamental matrix except it uses normalised coordinates, which have had the intrinsic matrix applied i.e. $s_0 = \boldsymbol{K}^{-1}s$ where $\boldsymbol{K}$ is the intrinsic matrix. A consequence of this is

$$\boldsymbol{E} = [\boldsymbol{T}]_x \boldsymbol{R} \tag{2.26}$$

where $\boldsymbol{E}$ is the essential matrix, $\boldsymbol{T}$ is the translation between the cameras and $\boldsymbol{R}$ is the rotation between them. The essential matrix can be calculated using the intrinsic matrices as[13]

$$\boldsymbol{E} = \boldsymbol{K}'^T \boldsymbol{F} \boldsymbol{K} \tag{2.27}$$

and then decomposed into the rotation and translation parts by decomposing $\boldsymbol{E}$ using SVD as $\boldsymbol{E} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{V}^T$ then

$$[\boldsymbol{T}]_x = \boldsymbol{U}\boldsymbol{R}_z\boldsymbol{D}\boldsymbol{U}^T \tag{2.28}$$

$$\boldsymbol{R} = \boldsymbol{U}\boldsymbol{R}_z^T\boldsymbol{V}^T \tag{2.29}$$

where $\boldsymbol{R}_z = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ or $\boldsymbol{R}_z = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix}$ [21]. The projection matrices can then be reconstructed as

$$\boldsymbol{P} = \boldsymbol{K} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{2.30}$$

$$\boldsymbol{P}' = \boldsymbol{K}' \left[\; \boldsymbol{R}^T \;\middle|\; -\boldsymbol{R}^T\boldsymbol{T} \; \right] \tag{2.31}$$

Note that scale can not be recovered.

2.8 Sumary

The previous sections have covered many of the algorithms used by this system. Specifically, NCC and the Harris corner detector given in section 2.2, Equation 2.15 is used to calculate 3D coordinates given image coordinates, Equation 2.23 to calculate the fundamental matrix and Equations 2.24 to 2.31 to calculate the projection matrices. The rest of this chapter has been concerned with the basis for the system and background to the following chapters.

[13] The SVD of $\boldsymbol{E} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{V}^T$ should have $\boldsymbol{D} = \begin{bmatrix} \alpha & 0 & 0 \\ 0 & \alpha & 0 \\ 0 & 0 & 0 \end{bmatrix}$. This should be enforced with $\alpha = (\boldsymbol{D}_{11} + \boldsymbol{D}_{22})/2$ before multiplying back to get $\boldsymbol{E}$.

3 Proccessing Models

3.1 Overview

The task of converting a set of 2D images into a 3D model is ultimately a series of processing steps, illustrated by Figure 3.1. The most important part of Figure 3.1

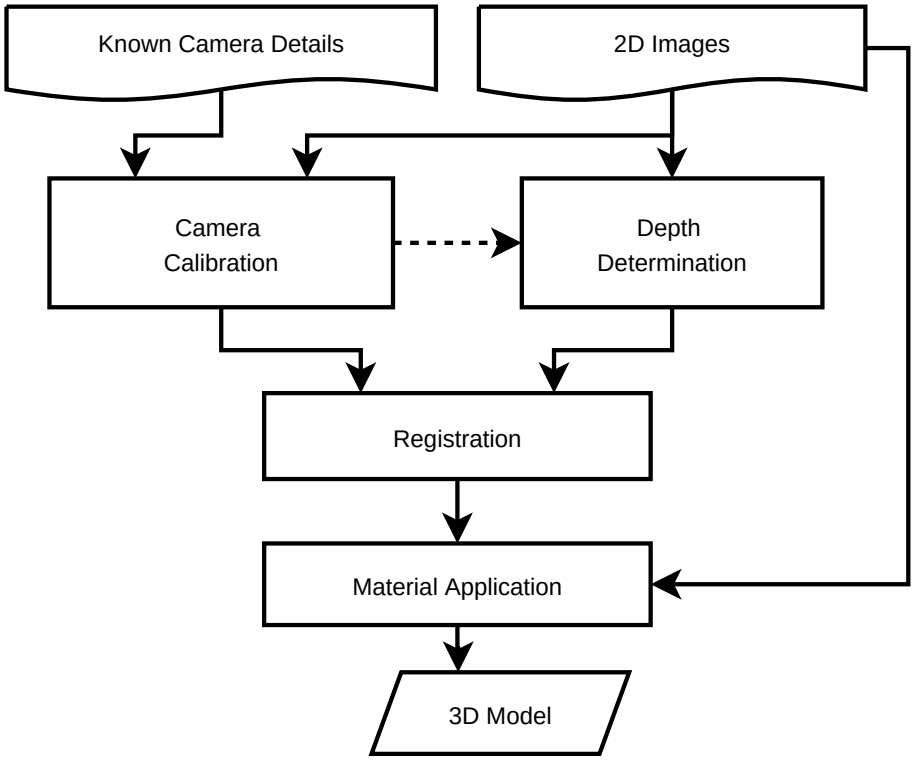


Figure 3.1: Data flow in a 2D to 3D solution. Primarily based on the solution presented by [5].

is depth determination, as this calculates the 3D component missing from any given image - depth. In fact, a fully functioning system can exist with *just* this component, assuming that camera calibration is known and there is no intention to move the virtual camera far from the real cameras position.

Without camera calibration at some level no arrangement of algorithms can work, this can range from simply knowing it through to calculating the calibration from the images at hand. The dotted line between Calibration and Depth

Determination represents that the camera calibration is usually required to determine depth, for instance steropsis requires that the epipolar geometry be known.

Once you have multiple depth maps you have to combine them to create a final mesh, this is the process of registration[1]. This uses camera calibration to identify where the many meshes created by depth maps can be joined together to create a larger mesh that stands up to observation from more than one angle.

By the stage of Material Application you have a complete 3D mesh, which may be the final goal, but usually you will want to apply the colour from the original photographs to the mesh. This can range from projecting the images onto the mesh arbitrarily, through methods of merging the textures for super-resolution[2] and ultimately to subdividing the mesh by material and inferring details about those materials, such as specular and diffuse properties.

The following sections cover each part of Figure 3.1 in turn, with a final section on other methods.

3.2 Camera Callibration

There are three common scenarios when it comes to camera calibration; known configuration, off-line configuration and self calibration. In all cases we are aiming to obtain the projection matrices of the cameras. For the following list it is assumed that multiple cameras are involved, if one camera is involved then only the intrinsic parameters need be recovered; this can be done using an off-line technique.

Known configuration simply means both the rotation and translation between cameras and the intrinsic matrices for cameras, and therefore their projection matrices, are known. This usually happens with a stereo rig where two cameras are fixed with no rotation and a translation perpendicular to there shared direction between them. If constructing a stereo rig the intrinsic parameters will be unknown and can be found using an off-line technique.

Off-line calibration is done in two stages, intrinsic parameters then extrinsic parameters. The first stage involves photographing a target with known parameters then using techniques depending on the type of target to determine the intrinsic parameters. Examples of this would be [7, 22]. The second stage is done on-line by finding matches between scene points, calculating the fundamental matrix and reconstructing the projection using the fundamental and intrinsic matrices. (See section 2.7) Off-line calibration can also be done on-line if there is a target to use in the scene, either put there or manually measured.

[1] Registration also refers to the post-processing of separately gathered depth maps and colour maps to line them up, often as a consequence of using a 3D scanner.

[2] Super resolution is the technique of merging photos of the same object from multiple angles at low resolution to create a higher resolution texture of that object resulting in more visible detail.

Self-calibration generally involves finding the fundamental matrix and reconstructing the projection matrices within a projective translation using the techniques discussed in section 2.7. The Projective Reconstruction Theorem[19] states that there is a transformation $\boldsymbol{H}$ such that

$$\boldsymbol{X}_2 = \boldsymbol{H}\boldsymbol{X}_1 \qquad \boldsymbol{P}_2 = \boldsymbol{P}_1\boldsymbol{H}^{-1} \qquad \boldsymbol{P}'_2 = \boldsymbol{P}'_1\boldsymbol{H}^{-1} \tag{3.1}$$

there are many possible $\boldsymbol{H}$ but only one results in the *correct* reconstruction. Finding the correct $\boldsymbol{H}$ involves applying scene constraints, such as parallel lines, angles between lines or known vertex positions. With three or more images self-calibration can be done with just the images at hand, using techniques such as the Kruppa equations[8].

3.3 Depth Determination

Excluding direct methods such as 3D scanners (2.4) depth can be determined for an image either by using just that image or multiple images.

3.3.1 Single Image

Generally, calculating depth from a single image is an ill-posed problem[17, p. 447], but by applying extra constraints it can be done. If the scene is constructed of known objects then those objects can be found and 3D geometry already obtained matched to the scene[23]. Such techniques are limited by the fact that only known objects can be reconstructed.

Visual cues can be used to provide the necessary information, such as focus, light and texture. If you can determine how far in/out of focus a point in an image is (A defocus operator[24].) then using a detailed camera model depth can be determined. There are numerous constraints on this technique, such as the detailed knowledge of the camera required and its dependence on a small depth of field. By modelling light and how it results in the particular pixel colours surface normals can be determined[25], this requires known lighting. Surface normals can also be discovered by examining how a regular texture deforms with the shape of an object. Both these techniques produce needle maps (A representation of surface orientation[3].) from which shape can be reconstructed.

3.3.2 Multiple Images

Stereopsis, using multiple images to calculate depth, is a well defined problem when compared to working with single images. The correspondence problem, that of finding matches between two images so the positions of the matched elements can then be triangulated in 3D space is at the core of this. There are

[3] http://www.bmva.ac.uk/bmvc/1997/papers/094/node1.html

two main categories of the correspondence problem, sparse and dense. Sparse can be further divided into wide and short baseline.

Dense stereo involves determining a depth value for every pixel in an image, or defining it as occluded. Nearly all such algorithms require rectified input. This is where a pair of images have been transformed so the epipolar lines are horizontal, limiting matching to 1D along the lines of the image, as in Figure 3.2. To rectify the input you need to know the epipolar relationship between the

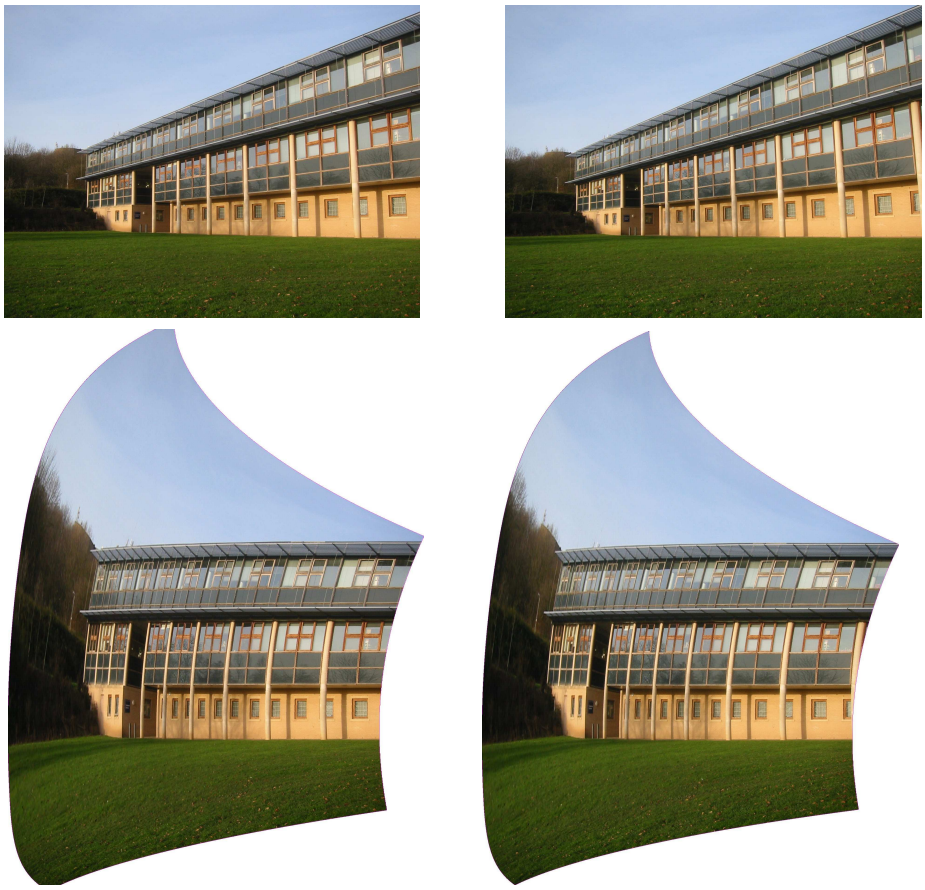


Figure 3.2: Original image pair followed by Rectified image pair, calculated using [26]. You can visually verify them by checking that the horizontal lines contain the same details.

images. This can be extracted from the fundamental matrix, which can be calculated using matches between images. (See Equation 2.23) Thus dense matching is often boot-strapped by sparse matching.

Sparse matching involves first extracting features, such as edges, corners or regions, then matching these features together between images, corners are usually used. Short baseline stereo is used to refer to techniques where the matching is done with no consideration for any transformation between the images, these will fail if there is too great a disparity between images. Wide baseline stereo

algorithms are designed specifically for the task and can cope with angles between views of 60° plus[27]. If used for depth determination sparse matching comes with the disadvantage that the sparse depth values have to be interpolated between features.

Once the correspondence of an image has been obtained a 3D mesh can then be produced, by calculating depth and projecting out of the camera. See Equation 2.15, which does this directly from correspondences.

3.4 Registration

In this context registration is taking the 3D data inferred by depth determination for each image then combining it to produce a 3D model that is larger than any single view. Following Figure 3.3 we start with multiple chunks of geometry. (a)

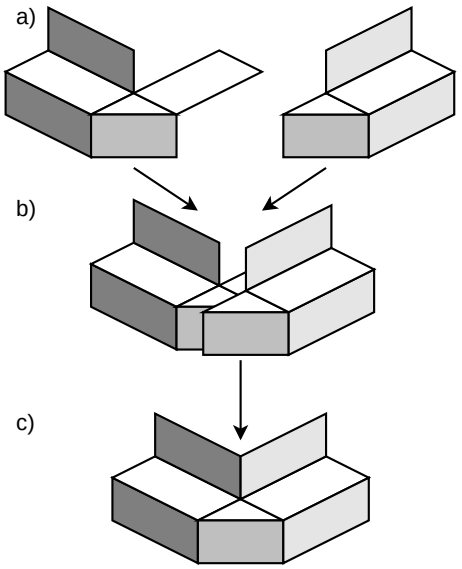


Figure 3.3: Registration

If transferred into the same coordinate space using the relative camera positions the geometry would not intercept accurately due to noise in the measurements, and shared geometry would not be deleted. (b) Registration is about merging the data such that these issues are resolved. (c)

Aligning the meshes is a matter of using the already detected correspondences between images and minimizing distances[28]. Shared geometry however is problematic, this is because the depth map is usually of a discontinuous surface so you cannot assume it is one large mesh. There are many basic approaches, such as vertex clouds, stitching and carving. With a vertex cloud you throw out any potential meshes implied by the continuity of the images and work only with the points as one data set, you then attempt to infer surfaces between them

from scratch. With stitching you segment the image to create a set of continuous surfaces, these are then stitched together. With carving you use the depth maps as definitions of empty space and consider everything else as solid, you then carve away the depth maps data from the solid space.

3.5 Material Application

At its simplest applying a material to the 3D model is applying a colour texture, this can be done by simply mapping the original images onto the parts of the 3D model created by those images. When multiple images map to the same section of geometry one of the images can be chosen arbitrarily to take priority or they can be blended. If blending is done it is often weighted in favour of images from cameras closer to the geometry and image projections closer to being flat to the geometry. Beyond this simple technique many others exist, such as:

- View Dependent Texture Mapping is blending the images depending on the *viewing* angle, such that images taken from a similar angle to the viewer take priority. This can be very effective as details too small to be captured by the 3D model can obtain a sense of depth on account of being rendered from the correct direction.
- Super resolution, for example [29], involves combining the images to create a higher resolution image.
- Material parameter modelling is going beyond simple colour and measuring the details that define how a material changes colour depending on viewing angle and viewing conditions. This requires many, many samples of the material and is impractical in a real world scene as controlled lighting is usually required. The many samples required can be obtained by segmenting the model by material, so you get many samples from many directions for each material.

3.6 Final Model

Once a final a 3D model with materials has been created it will then need post-processing. Noise is a substantial problem and needs to be minimised, this can be done using various statistical techniques to detect and smooth or delete outliers. Standard signal processing can be applied to meshes[30], alternatively outlier detection can be based on detecting significant deviation from neighbouring statistical properties. Once noise is removed holes then need to be filled in, interpolation techniques can be applied. For real time rendering model simplification and conversion to the correct format is often required. If detailed material information has been obtained a specific renderer will be required to visualise it.

3.7 Other Methods

This section contains a list of alternative system arrangements which do not fit within the above data flow model.

3.7.1 Homo-Sapian

It may seem strange to include the humble human being in a list of systems, however reality is that the human being is the best system available for this task. This particular *algorithm* can be applied to all parts of the problem, and given a 3D modelling package, a tape measure and time (And probably money.) will solve the entire problem with a high degree of accuracy. (Assuming the object(s) in question are of a suitable scale.)

On a more practical note, solutions that automate the process but allow a human to add there own input and correct the computers mistakes are inevitably more successful than systems that work alone, and form the most robust systems that can be currently created. (An example would be [31], a technique used in modern films[4].) The issues with implementing such a system are non-trivial however, as interfaces[5] and work flow becomes an issue[6].

Despite being the best solution, due to the time constraints human interaction with the process will no longer be considered as even a partially complete solution that has interactivity is unachievable in the time available.

3.7.2 Space Carving

The principal behind space carving is to take a large solid space and then for each view of the object calculate the outline of the object and carve away the parts that are outside the outline. If done from enough views this creates an accurate model of the object. This requires that views can be captured from all angles of the object and that the object is not concave. It also requires that the object can be distinguished from the background to find its outline, which usually requires the background be known, limiting such solutions to studio environments. Whilst limited in its capability the simplicity of this technique makes it very reliable.

[4] http://www.debevec.org/Campanile/

[5] Such interfaces are complete 3D modelling interfaces with extra functionality. An examination of any popular 3D modelling tool (Such as Maya, http://www.alias.com.) shows that many man years have gone into its design and implementation.

[6] The direct consequence of work flow being an issue is that algorithms that take more than a few seconds become inappropriate. In consequence the computer becomes less of a contributor as simpler algorithms are used.

3.7.3 Image Based Rendering

Image Based Rendering[7] never constructs a model of the scene, instead it directly constructs new views of the scene from the images. The motivation behind this is that a 3D model is usually constructed to be rendered back as another view of the scene, so you have the computer vision pipeline followed by a rendering pipeline. Instead of this the two pipelines can be merged and customised for the task. This comes with the advantage of producing a very realistic result as it works directly from real world data, it is also independent of scene complexity and can capture details that model based techniques lose. It does however limit the output to non-realtime behaviour as it can not use standard real-time rendering pipelines. It also requires many parts of the standard pipeline and it is of greater complexity to implement.

[7] http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/LIVATINO2/MainApprRVVS/node1.html

4 Design

Four sections follow, the first explains the development process, the second the requirements of the system. The final two sections are the designs for the framework (This is explained in the relevant section. To summarise a framework has been used to abstract between data flow and data processing.) and algorithms respectively.

4.1 Development Process

A variation on the incremental development process has been chosen, see [32, p. 51]. The incremental development process (A predecessor to extreme programming.) is defined as having a set of requirements divided between a set of increments, each cycle an increment is implemented and tested, until the system is complete, see Figure 4.1.

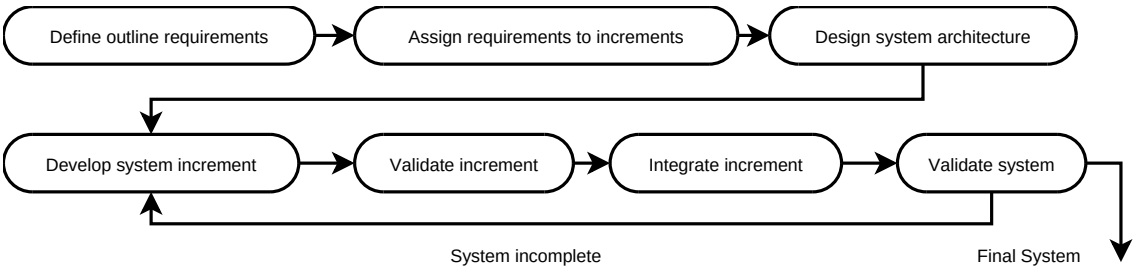


Figure 4.1: The design flow diagram for Incremental Development, copied directly from [32, p. 52]

This model was chosen as its incremental nature suits a modular system where the focus is not on a set of features but on improving a single feature. (The quality of the 3D output.) It has been modified to suit this project as there is no possibility to split requirements over increments, instead a phase has been added to assess how the greatest improvement can be obtained each cycle. The nature of the project means no final goal will be obtained, so instead the exit condition tests if enough time for improvements exists, if not the process stops and the system is considered complete, see Figure 4.2.

The architecture step is also taken to be an implementation phase in addition to a design phase in this variation. It is here that the framework is implemented before work on an actual solution starts. The first cycle is then concerned with

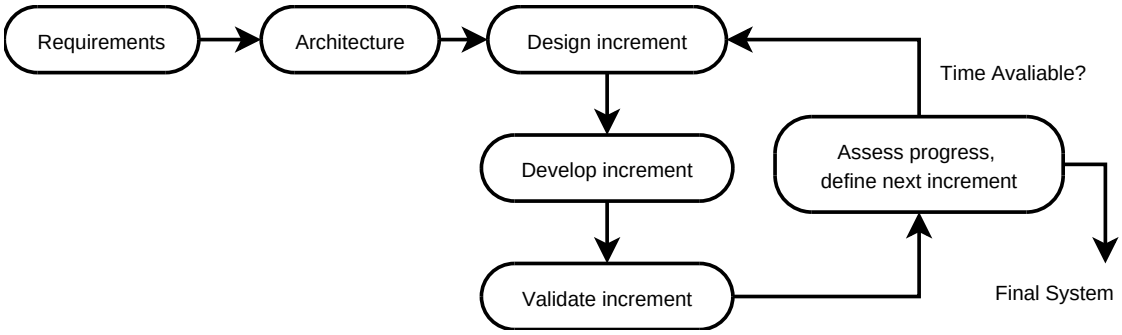


Figure 4.2: The variation on Figure 4.1 used for this project.

getting the most basic of working systems, all following cycles then improve on that, but at the end of each cycle the system must be working so work can stop if the exit condition is met.

4.2 Requirements

There are no defined customers for this system. Its purpose is to achieve a specific processing goal, so requirement engineering is concerned only with narrowing down the set of possible solutions. This means that requirements are ultimately concerned with narrowing the set of appropriate algorithms. A list of requirements follows, ordered by priority, highest priority first. Each requirement is listed with a rationale as to why it is specified and how it should be tested when appropriate[1].

- **3D Reconstruction of Scenes from multiple Photos**.
  The input is colour digital photographs taken from a consumer-grade digital camera. The output should be a navigable virtual representation of the scene recorded by the photographs, stored as a 3D model.

  Rationale: The primary requirement should be a working system. The input format has been chosen due to its availability from digital cameras, as most other types of sensor are expensive or hard to obtain. The output should be a 3D model. Obtaining a 3D model from photographs has seen the most research and working systems do exist[5], therefore the chances of success are probably improved.

[1] [32] has influenced this layout.

Testing: Comparison of the output of the system with the results of the best available vision system, a pair of human eyes. A quantitative analysis is not possible as the actual 3D models for captured scenes are unknown.

- **Casual Capture**.
  This implies that capturing the scene does not require special arrangements, such as a blue screen or specific lighting.

  Rationale: This is a follow on from the input being captured using a consumer-grade digital camera. By stating that the system must not apply such constraints in theory if development of the system was perpetual it should never reach the point where improvements would require a rewrite.

  Testing: If a constraint exists which fails this then it fails. Success is undefined however as there can always be constraints assumed without the realisation that they have been assumed.

- **Minimization of Scene Constraints**.
  Current solutions fail on many types of scene. Whilst it is unrealistic to be attempting to obtain state of the art capability there should be a preference towards robust and reliable algorithms.

  Rationale: Every capturing constraint reduces the number of scenes that can be processed by the system. Whilst some constraints can not be removed, for instance that the scene is static, there is no reason to not minimise constraints where possible within the time available.

  Testing: As the requirement states no specific constraints testing can not be quantitavely done. However, measurements can be made of the system limits and of what causes it to fail.

- **Automatic Processing**.
  The system should be automatic, in that human interaction with the system is to be avoided.

  Rationale: See subsection 3.7.1.

  Testing: None required. Note that this constraint has been broken for intrinsic camera calibration (4.4.3) due to time constraints.

- **Performance is not a priority**.
  Once a module is working time will not be spent making it run faster. Memory usage is also to be ignored, unless its more than the hardware available can manage.

Rationale: The time constraints on the project mean certain parts are going to have to take second place. Performance is an obvious candidate for this as its going to be slow anyway. Spending considerable time to take 10% off an hour of unattended proccessing is not justifiable.

Testing: -

- **High Modularity**.
  Any one part of the system should be replaceable, adding new features should involve minimal effort.

  Rationale: In the interest of current and future expandability a system should be modular. This should allow algorithms to be changed, or even base elements so it could be ported to run on a grid for instance.

  Testing: -

- **Development using `C++` under Windows^{tm}**.
  The system will be coded using `C++` to run on the Windows^{tm} platform, though an effort has been made to abstract all system calls. (See section A.2)

  Rationale: This is justified as it is the arrangement for which the most experience is available, therefore a certain class of problem are unlikely to occur.

  Testing: -

4.3 The Framework

Following on from the idea of treating 2D to 3D as a sequence of processing steps (chapter 3) the design has been split into two parts. The framework provides a script-driven environment, modular structure (Using dynamically linked libraries. [DLLs]) and a general purpose data type for inter-algorithm communication. The algorithms are a collection of modules that plug into the framework. Essentially the framework abstracts data flow from the algorithm implementations and allows changes to the data flow without recompilation. The other design possibility was to not have a framework, and to hard code the connections between algorithms. The advantages of a framework over none are:

- **Modular Development**. This allows algorithms to be developed without consideration for the rest of the system. Algorithms that are not performing as expected can be swapped out without any changes other than implementing the replacement. Also, the system proposed keeps the algorithms

in dynamic libraries that can be separately compiled[2].

- **Scripted Execution**. This allows the data flow to be arbitrarily edited. Algorithms can be swapped out without recompilation, or compared side by side. By having multiple script files with one executable it saves on compiling a different executable for each task the system might do. It also acts as an interface, so instead of the system asking for input it can be typed into the script file. This reduces development complexity and eliminates babysitting of the program. It also provides a form of precise documentation.

- **Single Variable Type**. [SVT] With only one data type available time does not need to be spent designing, implementing and debugging each data structure required. It also allows a script to save itself in an arbitrary state without developing savers/loaders for each data type. This is useful when developing modules far into the system, as you can run the system up to just before where you are working, save the state then load it in each time you run your new code instead if running it up to that point.[3]

and the disadvantages are:

- **Development time**. The time available is the primary restriction. Implementing a framework consumes time which could be spent on algorithms. Ultimately I believe that it will save time however, as many of the advantages mentioned above will reduce total work.

- **Single Variable Type**. [SVT] The design of this data type has to be perfect, otherwise it will be unable to pass all the types of data required between algorithms. Even then, having only one data type can involve passing data inefficiently and using nasty hacks to bypass design shortcomings. The data types passed between algorithms in this system are reasonably consistent however (Images and matrices), so this should not be an issue.

- **Speed**. On execution the system has to load and parse a script, it also has to load in modules. Once running the inefficiency of a single data type will slow things down. It is probably safe to say that such time delays are neglible in comparison with the algorithms that are run however.

Based on the above list the advantages of a framework are considered to outweigh the disadvantages, especially as the flexibility provided can be of great use when working within a short time span. Possible design alternatives for the

[2]To give an indication of the advantage this provides a complete compilation of the system at the end of development took 1:07 minutes, whilst recompilation after changing a single algorithm was 7 seconds.

[3]Stereo correspondence algorithms can take hours to run. Development of algorithms that run afterwards would be tedious without saving the results.

framework itself are listed in the following list, with reasons as to why these options were not taken:

- **Multiple Variable Types**. Instead of having a single variable type to contain all possible data types multiple types could be developed for each type of data required. This comes with the disadvantage of extended development to implement them. As the algorithms are modular the data types would sensibly then need to be modular, this would need an advanced saving/loading scheme with an object factory etc. The advantage is data structures that precisely match the requirements of algorithms. The extended development does not seem to justify this minimal advantage.
- **Extra Variable Types**. As a less extreme variant of the above a set of non-modular variable types could be provided. Whilst simple to implement its hard to find extra types that compliment the SVT as defined, see subsection 4.3.2.
- **Non-Modular Development**. Instead of developing each set of algorithms in separate DLLs they could all be compiled into a single executable. This makes compilation easier, and means dependencies are going to be less of an issue, but at the expense of compilation time. Ultimately compilation is done with greater frequency than makefile updates, so time should be saved by using DLLs.

The design now follows in terms of its interfaces, not its implementation, for the details of the implementation see Appendix A. It has two interfaces, the scripting language interface used by users of the system and covered in the first sub-section and the module interface for algorithms, i.e. the SVT, covered in the second section.

4.3.1 Scripting

Each algorithm takes a set of inputs and then produces a set of outputs, these inputs and outputs all being SVTs. The SVT is suitable for large data structures, but not for single numbers and other such details that are needed to define how an algorithm behaves. For this purpose extra details need to be passed into algorithms, though no requirement exists to pass them back out again or edit them within a script. To call an algorithm then requires the following data:

- Algorithm Identifier.
- A set of input SVTs.
- A set of output SVTs.
- Configuration data.

Any given script file is a sequence of algorithm calls, the nature of the processing means that no program control constructs (loops, ifs etc.) are needed. Variable types could either be declared or created on first use. If created on first use no scoping could exist, scoping provides advantages in that variables can be deleted when they go out of scope[4] instead of at the end of the program, also scripts written separately can be combined even if variable names clash. This means that variable declarations and a hierarchy structure are required.

The script files need to be edited without the creation of a custom editor, so text files are used. Instead of defining a custom format for the scripts XML[33] (Extensible Markup Language) has been used. Whilst this would probably be inappropriate for a normal scripting language, on account of its verbose nature, the lack of program control makes it suitable here. XML supports hierarchies and an element can be used to represent each algorithm invoked.

Instead of specifying the structure it will be exemplified by Figure 4.3. There

```
<op>
    <var name="varA"/>

    <op uses="varA">
      <var name="varB"/>
      <var name="varC"/>

      <op name="alg" in="varB,varC" out="varA"/>
    </op>

    <op name="save" in="varA" filename="results.txt"/>
</op>
```

Figure 4.3: Example of Scripting language

are two types of element used, `var` (Variable) and `op` (Operation). The `var` element declares a variable, its name given in the attribute '`name`'. The `op` element calls an operation, operations are a superset of algorithms, implemented in DLLs. DLLs can also supply operations to do non-algorithmic things, scoping would be an example. The `name` attribute given to the `op` element specifies which algorithm to call, it defaults to `exec` (execute), the scoping operation, which is why the first two `<op>`s work. The `in` attribute specifies the input variables and the `out` attribute specifies the output variables. These variables are internally accessed by the operation as an array, so order matters. Also given to the operation is the element of the Document Object Model[34] [DOM] that caused it

[4] By nature of processing images and 3D models the system has to process large amounts of data. For instance the script in subsection C.4.1 used over 1.5 gig of memory in its first iteration before scoping was added.

to be executed. This allows the operation to access any arbitrary parameters in that element, such at the `filename` attribute given to the `save` operation in the example. With some exceptions operations are named using a hierarchy, starting with the module name. For instance, '`math.matrix.mult`' is the name of an operation in '`math_matrix.dll`', which multiplies two matrices together.

This scripting language was chosen for its simplicity, both to implement and to explain. It was also chosen for its potential expandability, as whilst not required or even desirable operations could be written to implement loops, functions etc. All alternatives would be harder to explain and/or implement, except in regards to the scripting language. For instance a `C` style one could be chosen, however XML comes with the associated DOM. This defines the data structures for storing XML, so no design needs to be produced for that part of the system.

The final part of the user interface is the resulting program, which runs from the command line with a script file, it then loads the script, parses into a DOM and executes the root operation.

4.3.2 Single Variable Type

The SVT has to store many arrangements of data, so it must be structured to generalise all these possibilities. Structures it will have to store include:

- **Images**. These are represented as 2D arrays of colours or grey scale values.
- **Depth Maps**. Images with depth values associated with each pixel.
- **Matrices**. These are represented as 2D arrays of real values.
- **3D Models**. One of many representations is a set of objects including:
  - Images, identical to above, used to texture the model.
  - Vertices, a 1D array of positions, normals and texture coordinates.
  - Triangles, a 1D array of three indices to vertices.

The structure of the first three generalises to a n dimensional array of fields, where fields are integers, reals etc. The 3D model contains several of these combined to create its structure. This implies a hierarchy of nodes, where leaf nodes contain n dimensional arrays of specific data items. The SVT has been defined to match this model.

There are two types of node, branch nodes and leaf nodes. The branch nodes contain both branch and leaf nodes and exist to construct a hierarchy. The root node of a SVT is always a branch. The leaf nodes are n dimensional arrays of fields. There is an `enum` of field types, each of which has a defined type and therefore size. Any given element in the array can only have one of each field type. (Multiple are not required.) All this data is exposed by a set of accessors, see section A.4. See section A.6 for a list of variable types expressed using this

system. Various helper methods and an iterator class are provided, as otherwise they would be duplicated within many operations. Every node is assigned a type, this has no actual bearing except that each type has a set of requirements associated with it, so type checking is not about checking that a SVT contains the right number of dimensions and fields but that its type is equal to the one expected. A hierarchy of types was considered, so you could have a RGB image as a sub-type of image for instance, but considered too complicated for limited advantage.

There are alternate designs, such as

- The fields within each element of the leaf array could be defined on a per element basis instead of a per leaf basis.
- It could be implemented as a sparse array.
- Fields could be generalised, so arbitrary fields could be inserted into the arrays.
- In addition to the array structure each node could also have a set of properties in '`token=value`' form, to store arbitrary meta-data about the data in the node.

None of these alternatives confer any real advantage or provide features that are actually required, the solution proposed is considered the best solution as it is the simplest solution that provides all functionality and pushes no work into the algorithm implementations. (i.e. the leaf type could simply be a block of memory, but then each algorithm would have to index that block itself and would be doing work that should be centralised.)

4.4 The Algorithms

On account of the development process the algorithms have been split into a sequence of increments, the first increment is designed to get a bare minimum working system, all further increments were then designed to improve on it.

4.4.1 Increment 1, Stereo Correspondence

As previously covered in section 3.1 a working system can exist with just depth determination, for this reason the first iteration is to implement this. Specifically, stereo correspondence is the method used. This approach has been taken as it best matches the casual capture requirement, in that all other techniques require specialist tools or specific environments to work. Of all the possible solutions it also has had both the most research and success. As validation is required the iteration also requires enough supporting functionality to get the images

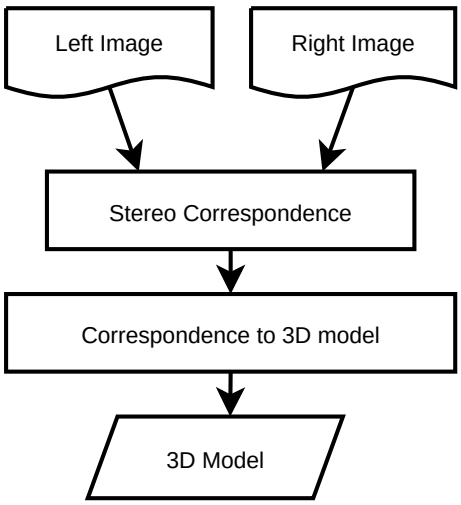


Figure 4.4: Data Flow Diagram for Iteration 1

into the system, convert the result to a 3D model and to output the result, as in Figure 4.4.

As only stereo correspondence is implemented the system is limited to image pairs where there is only horizontal translation between the two, i.e. a standard stereo rig. The lack of camera calibration also means that 3D coordinates can not be correctly reconstructed, instead the intrinsic parameters are assumed to be ideal[5]. To support the correspondence algorithm various operations are required

- Image loading.
- Correspondence to depth map.
- Depth map to 3D model.
- Noise reduction/cleaning up etc. of depth maps.
- Auxiliary modules for validation of results.

The following two sub-sections discuss the choice of stereo correspondence algorithm and output of the 3D model.

Correspondence

The correspondence problem in its many forms (subsection 3.3.2) is probably the most studied problem in computer vision, there are more algorithms than can

[5]Specifically, assumed to be $\begin{bmatrix} 2 & 0 & 0 \\ 0 & 2\frac{2}{3} & 0 \\ 0 & 0 & 1 \end{bmatrix}$. This is based on the fact that the image has been normalised from a 4:3 aspect ratio to [-1,1]x[-1,1] and assuming the field of view is 90°.

be sensibly reviewed. A good start is [35], a league table of stereo algorithms by accuracy. However, the top algorithms are complicated, and would take too long to implement. For instance, second place on the list[36] requires an understanding of four of its references before it can be implemented.

Instead *A Maximum Likelihood Stereo Algorithm*[37] was used for the following reasons

- It is used by [5], a working solution, so it should be good enough for the task in question.
- It is relatively simple, understanding and implementing it within the time available is a reasonable undertaking.
- There are improvements to the base algorithm[38, 39, 5], so it can be improved to the standard required.

The base variant of this algorithm uses dynamic programming on maximum likelihood derived costs of either matching features or marking them as occluded. It assumes *monotonic ordering*[6]. The base algorithm uses the luminance value of individual pixels as its features. Many variations of this algorithm were implemented, see section B.8, and extensive testing was done, see subsection 5.1.2.

Model Output

Once a 3D model is produced it needs to be viewed. There are two ways to view a model, either save it in a standard file format and open it in an external viewer or develop a viewer specifically for the task. Initially the first of these options was presumed to require the least amount of work, however an investigation of free model viewers showed a limited set of supported file formats, and of these formats documentation on them was limited. Of the formats that could be implemented several could not store the required data, the remaining few would of taken considerable time to get working. It was therefore decided that implementing a viewer would be easier, as OpenGL[7] is a known quantity.

The viewer was designed to be both simple and quick to implement. It is implemented as an operation, which is given the 3D model SVT defined in A.6.2, it then shows a window with the model in and pauses execution of the script till the window is closed. Browsing the model consists of dollying around a centre point in the scene by dragging with the LMB, panning when using the RMB and zooming with the mouse wheel. The centre point can be altered using the keyboard. This is a clone of the interface design common among 3D programs[8]

[6] $\forall i, j, u, v \bullet \text{matched}(x_i, y_j) \land u > 0 \land v > 0 \Rightarrow \neg\text{matched}(x_{i+u}, y_{j-v})$ where x_i is an indexed pixel in one image and y_j is an indexed pixel in the other.

[7] tm, http://opengl.org

[8] Such as Blender (http://www.blender3d.org) or Maya (http://www.alias.com).

without the need to get into/out of viewing mode, as it is the only mode available. Cloning a well known design has the advantage of familiarity, designing a unique interface confers no advantage in this case.

4.4.2 Increment 2, Rectification

The solution at this point had two issues, that it required a stereo pair and that it produced an incorrect 3D model due to the lack of camera calibration. Removing either constraint has no significant advantage over the other. Both are dependent on the fundamental matrix being calculated; the result of removing the stereo constraint is better suited for testing the fundamental matrix calculation than camera calibration so the stereo pair constraint was chosen. In addition, implementing camera calibration is a larger task, the chosen task is of a more reasonable size for a single increment.

Removing the stereo pair constraint requires rectifying the input to make the correspondence search 1D. Rectification requires the fundamental matrix, which can be calculated by matching corners between image pairs, this extends the data flow diagram as in Figure 4.5.

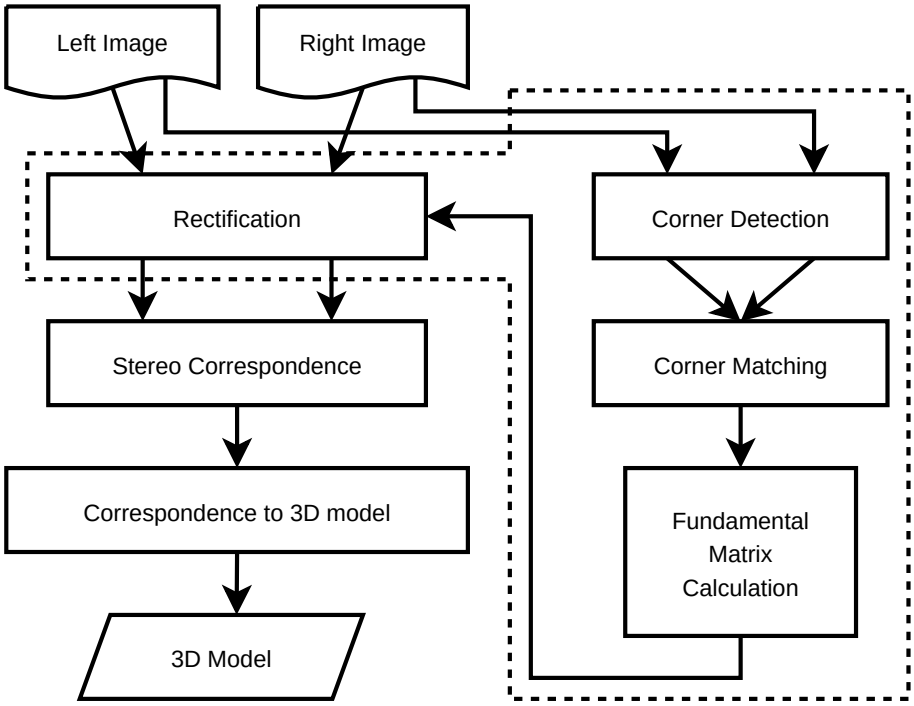


Figure 4.5: Data Flow Diagram for Iteration 2

The following two sub-sections discuss the fundamental matrix calculation technique and rectification algorithm.

Fundamental Matrix Calculation

Given a set of constraints between the two images the fundamental matrix can be calculated, these constraints are usually in the form of matches between corners so Equation 2.23 can be used. Corners have to first be found, the Harris corner detector (See 2.2.1) is used for this purpose. The Harris detector has been chosen as it produces the lowest rate of errors according to [40]. Matching corners is then done using NCC. More advanced methods that are capable of handling large disparities between images exist (i.e. [27]), however the rest of the system will not become robust enough to manage such scenes in the time available so there is little point in implementing anything more advanced.

The matching process just mentioned produces a sparse set of correspondences between two images. This set contains errors however, so SVD can not be directly used to calculate the fundamental matrix from them, instead a method known as RANSAC is used. Random Sample Consensus[41] [RANSAC] works on any model fitting problem where a large number of outliers are present. It repeatedly selects a random subset of the samples from which it constructs the model, it then checks how many other samples fit the model selected, these are that models inliers. After doing this many times the correct model is considered to be the one which obtained the most inliers. For our purposes this means

- Repeat until the probability of having the correct solution is high enough.
  - Select 8 matches randomly.
  - Calculate the fundamental matrix using those 8 matches.
  - Iterate all matches and count how many of them fit the epipolar geometry specified by the fundamental matrix. This is done by checking that the distance from the epipolar line calculated from one point is within a threshold of the other point of a proposed match.
- Use SVD on all the inliers selected by the iteration that got the most inliers and calculate the final fundamental matrix.

Rectification

Three rectification algorithms were examined, one representative of the traditional solution[42] and two more recent solutions[43, 26]. Pollefeys[26] points out that traditional solutions fail under forward motion, (When the epipole is in the image.) and whilst there is little between Polar Rectification[26] and Cylindrical Rectification[43] Pollefeys claims his is easier to implement. Therefore Pollefeys algorithm was chosen.

4.4.3 Increment 3, Camera Calibration

At this point the solution so far still had the camera calibration problem, but it also now had an issue with being restricted to only processing two images for generating the 3D model. Processing more than two images has great advantages for noise reduction. Adding many image support can be considered in two steps, firstly merging data for a single view, then merging multiple views to create large models. (Registration.) The second phase requires camera calibration, the first phase seems somewhat pointless when the output will be wrong because camera calibration has not been implemented. Camera calibration was pursued this iteration.

There are two approaches that could of been taken for camera calibration, off-line or self. (See section 3.2) Self calibration methods usually require three or more cameras, methods that work with less require extra data and would need an interface created for this purpose in violation of the requirements. Off-line on the other hand requires a separate calibration step to be done on a per camera basis with a specific object, a calibration target which must be created. As techniques work with calibration targets that can be printed this last point is not a problem. Having to calibrate each camera does come with the requirement that only known cameras be used, this restriction means that only data captured specifically for this system is likely to be of use. The limited capability of the system means this is the case anyway, Off-line calibration has therefore been used.

The requirement that no specialist equipment be required means that the older camera calibration techniques are unsuitable as they require 3D targets and/or specific camera motion. *A Flexible New Technique for Camera Calibration*[7] was chosen as it uses planar targets (that can be printed) and arbitrary motion, though only the basic approach in the paper was implemented. It is considered a good choice in part because its basic solution only requires a working SVD implementation, which is already required by other parts of the system. More advanced techniques exist (i.e. [22]), however they offer insignificant improvements for the increased complexity of there implementation. Implementing camera calibration has no effect on the current framework except that instead of using a guess of the intrinsic parameters they are calculated using this process, which is illustrated by Figure 4.6.
The following sub-section discusses finding target features.

Target Search

Given each view of the target (Figure 4.7) the corners of the black squares need to be found accurately. The corner detector from calculating matches between images can be reused to find all the corners in question, but identifying which once are which is then a problem. Initially, an attempt at automating this process was made, however it was taking too long to implement so loses were cut and

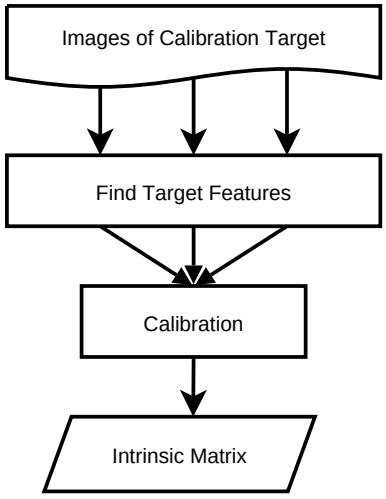


Figure 4.6: Intrinsic Camera Calibration Data flow diagram for Iteration 3

an interface constructed by which a human could select the corners. This runs contrary to the requirements, but as it was an off-line process and time meant an automatic system could not be implemented this solution was deemed acceptable.

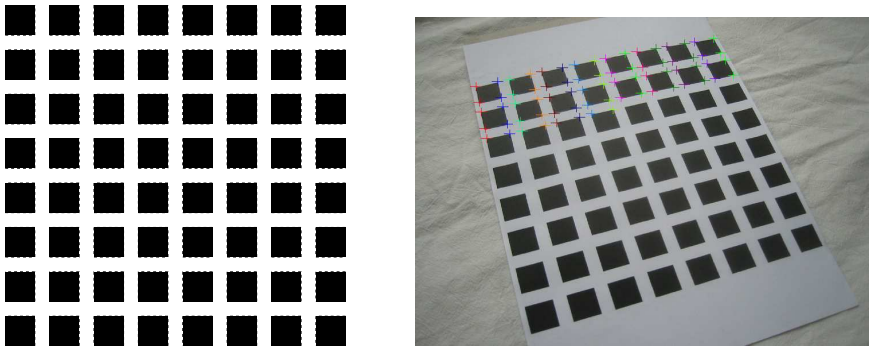


Figure 4.7: Camera Calibration Target on the Left, Photo of it on the Right

5 System Evaluation

This chapter has been divided into two, the first section focuses on validating that the implementation is correct at each stage, the second focuses on determining the limits of the system as an assessment of how it handles the requirements.

5.1 Validation

This section contains a sub-section for validating each step of the implementation, in line with the design process.

5.1.1 Framework

Minimal testing was done of the framework on the grounds that it was going to be tested every time it was run for the rest of development. The framework loads an XML file into a DOM and then runs the operations specified by the DOM; these two steps were tested separately. XML parsing was tested with a series of XML files known to be either correct or not, see section C.1. All correct files parsed correctly, all bad files showed error messages, except for test 3 which ignored the erroneous data and test 9 which ignored the second root node. These problems were ignored as they are not major. Note that this test only covers a few errors and is by no means thorough.

To test that the framework could load modules and execute operations within them modules were needed, so `var.dll` and `image_io.dll` were implemented, see sections B.2 and B.3 respectively for details. A variety of scripts were then run, see section C.2. The tests have the following purposes:

- **1:** Runs both of the built in operations, testing that operations in general work and both the operations in question work.
- **2:** Declares a variable to test SVTs and uses external algorithms to test module loading. Consequently tests image loading and saving.
- **3:** Tests scope.
- **4:** Tests variable saving and loading.

All tests produce there expected results, which also shows that the modules in question work.

5.1.2 Increment 1, Stereo Correspondence

There are ten stereo correspondence algorithms implemented, all variations of the same technique[37]. To test them four stereo pairs were used, three computer generated and one photographed, see Figure 5.1. The virtual scenes were rendered using POV Ray[1], see subsection C.3.1 for the source.
The algorithms are as follows

MLM	The first algorithm given in [37], using individual pixel luminance as the feature.
MLMH	The second algorithm.
MLMH+V	The third.
Colour MLM	The first algorithm but using colour to form a RGB feature vector. This takes the Gaussian distribution used for the maximum likelihood calculation in MLM and replaces it with the multivariate on the three colour components. Requires a covariance matrix instead of the standard deviation used by the base versions.
Colour MLMH	The second algorithm with colour.
Colour MLMH+V	The third algorithm with colour.
3X3 MLM	The first algorithm, working with a 3x3 window, with each of the 9 components forming a feature vector. Implemented using the same principles as the colour variant.
MLM NCC	The first algorithm, using Normalised Cross Correlation over a window to express a single valued feature. This is implemented exactly as base MLM, except instead of using the difference between pixel luminance it uses $255(1 - \text{NCC})$.
MLMH NCC	The second algorithm with NCC.
MLMH+V NCC	The third algorithm with NCC.

To test each algorithm was run on each of the pairs, in addition the algorithms using NCC were run with windows of 3X3, 5X5 and 7X7. In all cases correspondence maps were produced and qualitatively[2] analysed. In addition Table 5.1 was produced, indicating how long algorithms took to run and what percentage of each correspondence map was occluded/unknown. The timing is useful when deciding if an algorithm is worth the time required for the results it gives, the value of the occlusion percentages for analysis is debatable, but large differences between algorithms give a good indication that something is wrong. The three computer generated pairs are 800x600 pixels whilst the real world pair is 1600x1200, this is why the timing is different for I4[3]. The times were measured

[1] http://www.povray.org/

[2] A quantitative analysis could not be produced as the actual correspondence maps are unknown. For the POV Ray scenes they could plausibly be generated, however this would of taken considerable time to investigate and implement.

[3] The algorithm is $O(\text{width}^2\text{height})$ in regards to time.

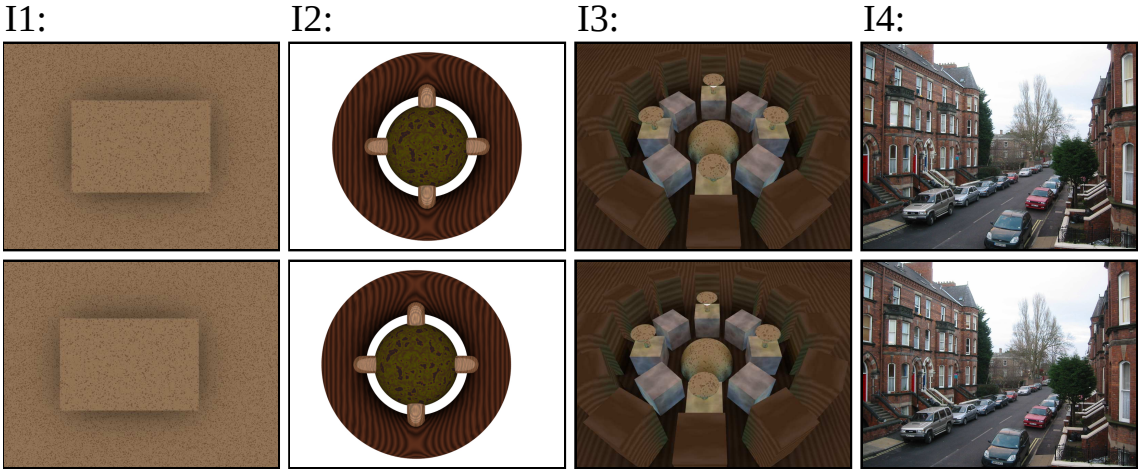


Figure 5.1: The test stereo pairs.

Algorithm	Time to Run I1/I2/I3/I4	Occlusion Percentage I1/I2/I3/I4
MLM	25s/24s/24s/3:02m	0.1/ 2.6/1.2/ 0.3
MLMH	37s/34s/35s/4:18m	0.4/ 2.9/2.3/ 0.5
MLMH+V	1:01m/56s/57s/7:55m	0.5/ 2.5/1.6/ 0.5
Colour MLM	47s/44s/45s/6:20m	0.0/ 2.3/0.3/ 0.1
Colour MLMH	1:01m/57s/58s/8:17m	0.2/ 3.4/2.3/ 0.4
Colour MLMH+V	1:29m/1:22m/1:25m/11:37m	0.2/ 2.7/1.8/ 0.3
3X3 MLM	5:57m/5:45m/5:58m/47:51m	1.4/ 0.4/0.1/ 0
3X3 MLM NCC	2:07m/1:55m/2:06m/24:56m	1.2/45.5/2.1/ 6.6
3X3 MLMH NCC	2:21m/2:04m/2:21m/28:31m	1.7/46.1/2.5/ 6.8
3X3 MLMH+V NCC	2:52m/2:33m/3:01m/34:36m	1.7/46.1/2.7/ 7.0
5X5 MLM NCC	3:41m/3:33m/3:43m/39:37m	3.1/44.9/2.7/ 9.0
5X5 MLMH NCC	4:05m/3:57m/4:10m/42:48m	2.7/44.8/2.9/ 9.7
5X5 MLMH+V NCC	4:53m/4:51m/5:05m/48:32m	3.1/44.8/3.2/10.9
7X7 MLM NCC	6:00m/5:46m/6:04m/59:50m	3.2/43.0/2.6/13.0
7X7 MLMH NCC	6:51m/6:43m/6:49m/1:01:32h	3.4/43.4/3.1/14.8
7X7 MLMH+V NCC	7:58m/7:49m/7:49m/1:09:04h	3.5/43.5/3.4/16.7

Table 5.1: Correspondence Algorithms time to run and percentage coverage.

with user time[4] and include only the correspondence algorithm. The Colour algorithms and 3X3 non-NCC algorithm have extra phases to calculate suitable covariance matrices, these are not included in the timing. (The extra phases take no more than a few second however.)

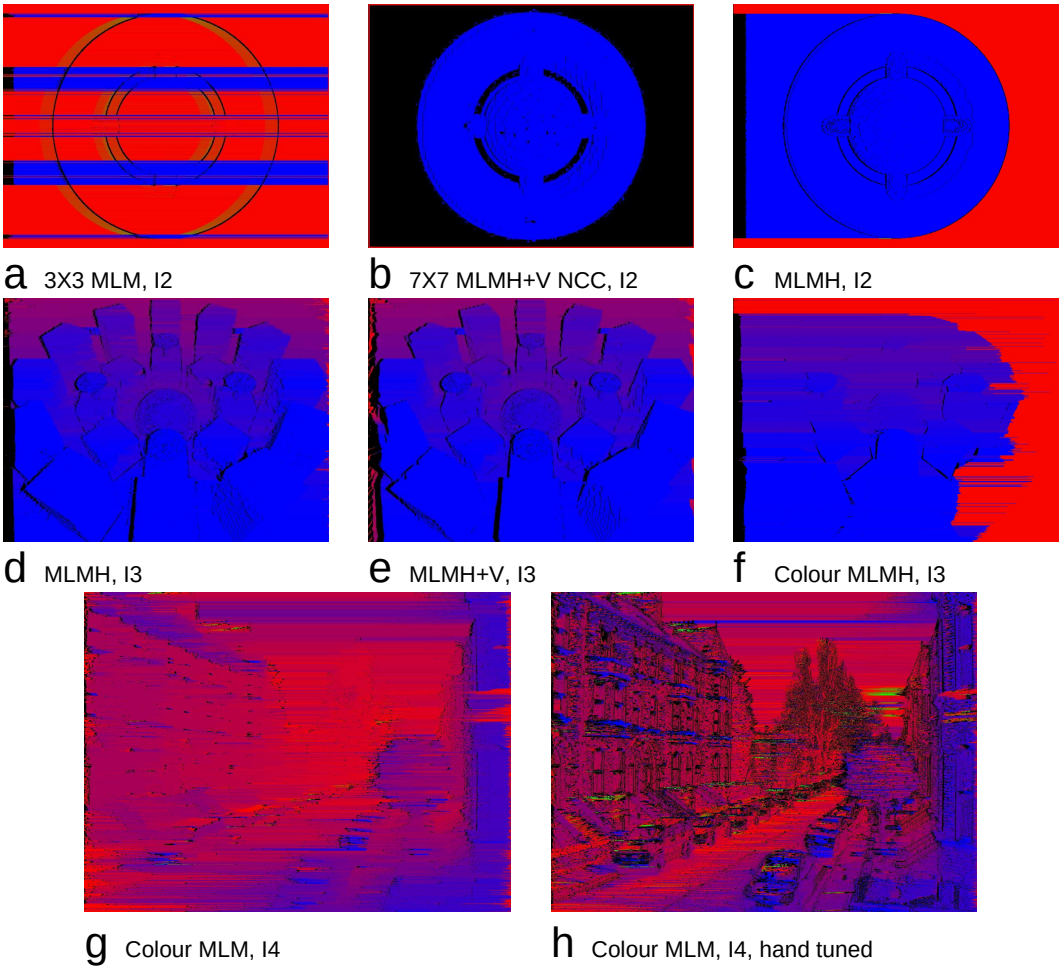


Figure 5.2: Various correspondence results referenced from the text

3X3 MLM fails on all pairs except I1, Figure 5.2,a[5] shows it failing on I2, whilst 5.2,b shows a *correct* result. 5.2,b was calculated using *7X7 MLMH+V NCC* and when compared to 5.2,c as created by *MLMH* illustrates the biggest difference between the first seven algorithms and the NCC based algorithms, the NCC

[4] As opposed to system time (operating system calls) or real time (other processes included). Should only include the CPU time used running the algorithm. Run on an Athlon^{tm} XP 2600+ (2Ghz) with 1Gb of RAM.

[5] All correspondence images are of the left image of a pair. Fading towards green from red indicates correspondence to the right, fading to blue from red indicates to the left and red itself indicates no offset. Black indicates no match, i.e. an occluded pixel.

algorithms mark background areas as occluded whilst the rest attempt to match, producing incorrect results. This can be seen in the change from around 2% to around 45% occluded in Table 5.1 for I2.

There is no major difference between the three variations of the base algorithm, whichever variant is used. There is an improvement from *MLM* to *MLMH* in that edges tend to be sharper. From *MLMH* to *MLMH+V* the quality gets worse in most cases. For instance from *MLMH* generated 5.2,d to *MLMH+V* generated 5.2,e the left edge loses all sharpness. Considering the evidence *MLMH* has to be the recommended class of algorithm, as *MLMH+V* consistently takes longer and does not appear to improve results.

The colour variants sometimes fail, for instance 5.2,f was produced with *Colour MLMH*, they are also capable of producing usable results such as 5.2,g, generated with *Colour MLM*. The problem appears to be that the covariance matrix is not suitably calculated, if hand tuned (Trial and error) good results can be obtained, such as 5.2,h., which is also *Colour MLM*. Even with an optimal covariance matrix the results do not surpass other variants. It seems reasonable to presume that *3X3 MLM* is affected by the same issue, but it would take too long to tune a 9X9 covariance matrix to find out.

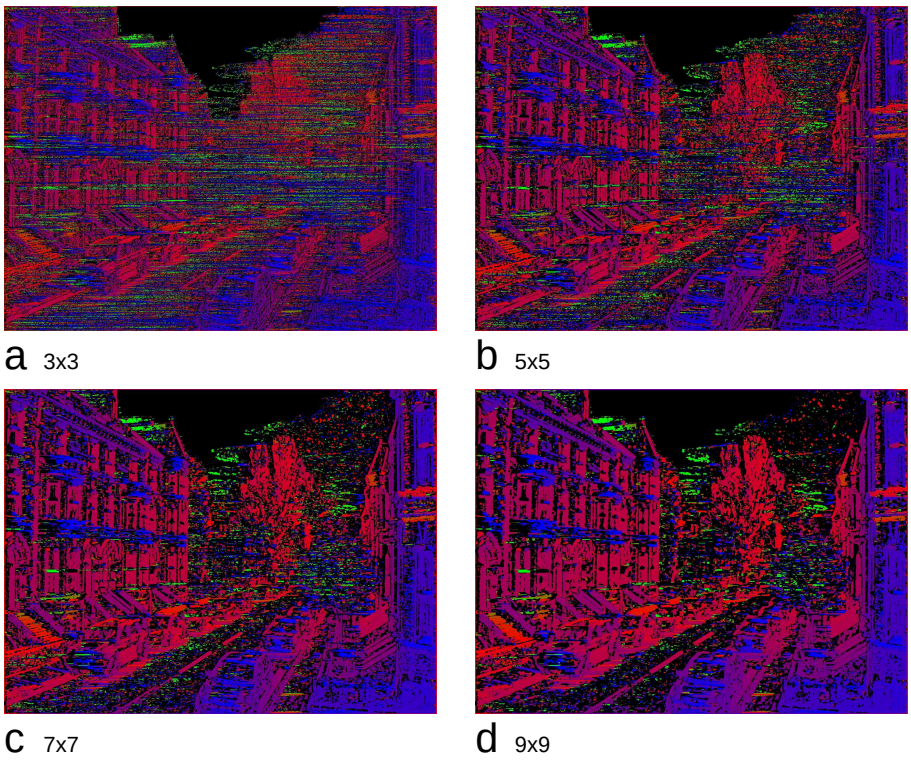


Figure 5.3: Various correspondence results referenced from the text

The NCC algorithms improve as the window size is increased. Figure 5.3 uses *MLM NCC* with a window size of 3X3 for a, 5X5 for b, 7X7 for c and 9X9 for d. As you progress through the images noise and errors are reduced. (This is especially apparent in the sky and on the road.) The NCC algorithms have a far greater percentage of occlusions, this is a problem as it makes noise harder to identify and means fine details will never be captured. Whilst the basic algorithms provide more data than NCC there inability to fail gracefully with backgrounds and noise is a greater problem. The other algorithms fail in some/all cases. Considering the algorithms available *5X5 MLMH NCC* is the sensible choice. NCC algorithms with larger windows produce better results but in general take too long.

5.1.3 Increment 2, Rectification

Rectification aligns features in both output images such that they are on the same horizontal line, so verifying rectification can be achieved by displaying the output images side by side and checking that this is the case. The algorithm used, Polar Rectification[26], requires special cases depending on the location of the two epipoles, specifically for both epipoles being in the image, one epipole being in the image and another outside the image and a final case for both epipoles being outside the image. The first and third case given are relatively easy to obtain (Forward translation and sideways translation between photos, respectively.), finding cases where the second scenario occurs is more a matter of luck than anything else. To test this 21 photos of a test scene were taken, code was then written (See subsection C.4.1) to generate a script that would calculate the fundamental matrices and rectify all image pairs in this set. The script outputted the epipolar coordinates for each pair so each of these three cases could then be found in the set, see subsection C.4.2 for examples of these cases. The results appear correct.

5.1.4 Increment 3, Camera Calibration

The output of camera calibration, the intrinsic matrix, can not be directly tested as its real value is unknown. However, its use should result in a reconstruction that is Euclidean. If Euclidean properties of the scene are known the reconstruction can be compared, the obvious property to test is angles, as distance would have to be compared by ratio due to the unknown scaling factor. Therefore testing can be done by performing reconstruction with the guessed intrinsic matrix and with the calculated intrinsic matrix. The calculated matrix should produce results where angles are nearer there true value. This is less a test as to if intrinsic calibration works but a test to determine if it improves the output of the system as a whole.

Figure 5.4 shows the aforementioned comparison, but both angles are the

Figure 5.4: Comparison of angles. There should be a difference in angle between un-calibrated, on the left, and calibrated, on the right. Both angles are about 130° though. (They should be 90°.)

same. The camera calibration therefore produces results no better than if calculated from the cameras details. This is presumed to be due to using a simple calibration technique, with a more advanced technique producing better results. Even so, this result is worse than expected.

5.2 Capability

This section first analyses the limits of the matching system, it then discusses failure scenarios.

5.2.1 Matching Limits

The limiting factor in the system is the matching of scene points between different images. Baumberg[27] tests his proposed feature matcher by placing a target on a turntable, taking a sequence of photos whilst turning the turntable between each photo and then matching each image with the first image taken, so the angle between matching attempts increases. The same test was conducted here. A scene was placed on a turntable and a sequence of photos were taken with 1° of rotation between each photo. Each photo was matched with the 0° photo and the number of inliers as determined by the RANSAC fundamental matrix calculation recorded. The inliers were also outputted graphically so there validity could be checked.

Figure 5.5 shows the results of this test. The inlier count given is the number of inliers according to the RANSAC step whilst the bad inliers are those which are obviously bad from examination. By 10° the results are unusable, with the first errors creeping in at just 4°. Compared to Baumberg[27], who obtains reliable matching at 60°, this is useless. These numbers reflect experience of the

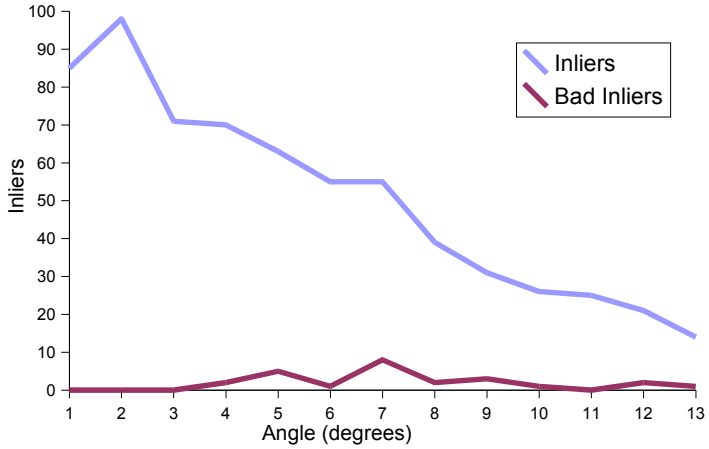


Figure 5.5: Graph of good matches found against angle around scene centre.

system, where a tripod has been required and care taken to avoid rotating it. See section C.5 for scripts and images related to this test.

5.2.2 Failure

There are many scenarios that cause varying levels of failure:

- **Planar Matches**. If all the matches are on a single plane then the fundamental matrix calculation is degenerate, and the output of the system is restricted to a plane.
- **Few Matches**. If the scene is too simple not enough matches will be found and calculation of the fundamental matrix will fail.
- **Low Textural Detail**. Areas with too little textural detail do not get matched. The standard stereo correspondence algorithms distort such areas whilst NCC based algorithms set them as occluded or match noise.
- **High Geometric Detail**. Areas with too much geometric detail, for example trees, get drowned out in noise.
- **Shiny/Transparent Materials**. Any material that significantly changes its colour with changes in angle causes correspondence to fail. This is especially bad with reflective surfaces, or even just specular highlights, as it matches the reflected detail causing large distortions in the reconstruction of the surface.

6 Conclusion

This section is divided into two, a critical review of what has been achieved and then a review of what could be obtained with further work, specifically with an eye to resolving the issues raised by the critical review.

6.1 Review

The system implemented follows Figure 4.5, each step has been verified to work individually and all the steps work as a group to successfully produced 3D models based on photographs. However, the scenes for which it will do this are limited and the quality of the output is poor. Figure 6.1 is the best output obtained, it has no real-world use due to the noise, failure to handle shiny objects and the limited range that can be obtained with only two photographs. Whilst not evident in the figure it also suffers from a projective distortion as the intrinsic matrix is not perfect (The virtual angle between the two walls of the house is about 130°.) and the walls suffer from noise close up. The following is a summary of problems observed throughout the results (with no consideration as to if the problems given *can* be solved)

- **Noise**. The noise produced by correspondence makes the models unusable. A heavy averaging filter has to be applied to reduce it to the levels seen in Figure 6.1, so all small details are lost.
- **Pairs Only**. The system will only take two images as input, this restricts the output range to that which is visible in both images. It also provides only one sample of each points position when multiple would assist noise reduction.
- **Poor Matching**. The matching algorithm is incapable of handling any real change in the angle between photos and can not manage much translation either. It is directly responsible for most failures due to it only matching corners on a single plane or making too many mistakes. This limits capturing to scenes with plenty of matchable objects, which generally means inorganic with lots of texturing information. The photos also have to be taken from almost the same angle, which can be hard to do without a tripod.
- **Shiny and Transparent Surfaces**. Surfaces that do not look roughly the same from every angle fail to match, causing dramatic distortions such as

Figure 6.1: Top image is of a house from an arbitrary view. It was generated using the stereo pair at the bottom.

the cars and windows in Figure 6.1. In addition, untextured surfaces tend to be distorted as there is nothing to match.

- **Distortion**. The intrinsic camera calibration is not very accurate, and the lack of compensation for barrel distortion does not help either. This results in a reconstruction that is wrong by a projective transformation, so angles and lengths are meaningless.

- **Materials**. Currently the photo is simply projected onto the model, no effort is made to merge multiple photos or calculate material properties. Renderings are therefore incorrectly coloured.

- **Movement**. The scene must remain static between photos.

Whilst the capability of the solution is limited the framework has proved to be a versatile tool, supporting modular development, fast experimentation and extensive testing.

6.2 Further Work

There are several improvements that can be made to the system by simply swapping out current algorithms and using better once:

- **Better matching**. The limiting factor of the current solution is matching, therefore the first piece of further work is to improve this part of the system. An algorithm such as [27] would be suggested as it can express the same interface as the current matching algorithm and be swapped in with no effort beyond implementation. It claims to work with angles of separation between photos of 60° plus, with such flexibility multiple images becomes a sensible proposition.

- **Multiple images**. Support for multiple images comes in two phases. The first is to merge multiple correspondence maps between one view and several others to generate a single depth map. This should reduce noise to manageable levels as multiple samples will now exist for each point and a higher percentage of the image should be matched. The second step is to add registration, to combine multiple depth maps into 3D models that cover an area larger than any one photograph. With these two improvements the first three issues listed in section 6.1 should be resolved.

- **Correct Calibration**. Improvements can be made to the intrinsic camera calibration to reduce distortion, the obvious approach would be to implement the more advanced methods from the paper used[7]. Whilst this would improve the results off-line calibration does not sit well with an automated system. Once three or more images become involved self-calibration

using a technique such as the Kruppa equations[8] becomes an option and would remove the off-line calibration step altogether. This should not only be more convenient but it should produce better results[1].

- **Improved Correspondence**. The current stereo correspondence algorithms are slow and unreliable, they were chosen primarily due to time constraints. Implementing a better one, such as any from [35], would improve the system. A faster one would also be recommended, as with multiple images being matched 42 minutes per pair becomes unreasonable.

- **Automated Pairing**. The current system requires the user to indicate which pairs of images should be matched. Automating this is possible[44].

- **Materials**. The system has focused on producing the correct shape, not on the correct material for that shape. There are many approaches that can be taken, as discussed in section 3.5. The result of using any should be a more realistic and more convincing rendering.

- **Compression**. The framework currently saves SVTs to disk without any compression, this results in enormous files[2]. Compressing them would save considerable disk space, especially as the above improvements would require the system to handle more data than at present.

Once the current system is both reliable enough and flexible enough in regards to its current functionality further improvements can be considered:

- **Managing Motion**. Whilst the problem of solving with motion between captured photos of a scene is ill-posed there is the possibility of domain-specific solutions. A possible approach is to know which objects will cause problems, find them, and remove them. For instance, if attempting to capture a town centre finding and removing people from the images before calculating correspondence. This would necessitate correspondence algorithms that can manage such gaps. If the objects are to be kept then matching a known 3D model to the images then using that model instead of calculating from the data can be done. There is then the possibility of using the outlines of shadows generated by these known objects to assist with determining geometry.

- **Inverse Lighting**. If material data is suitably captured then determining the positions of light sources should be theoretically possible. This can potentially lead to correctly integrating foreign 3D models into the data.

[1] This is because the intrinsic matrix changes as the camera adjusts to the scene in front of it. (See section 2.5) The calibration target will probably be shot up close to fill the shot whilst real scenes will be shot at a distance, so whilst the intrinsic matrix is likely to be consistent within these two sets its unlikely to remain the same between them.

[2] The directory containing all the testing data has over 5 gigabytes in it.

- **Instancing**. Once a 3D model has been obtained you will typically want to create arbitrary renderings of it. Finding and instancing duplicate objects, materials and textures would improve the response of a real-time renderer. The details that make each object unique could then be optionally rendered depending on resources available, or removed to save space. Each instanced *thing* would then have multiple instances, combining these instances could improve the quality of the generic *thing*.

- **Object Replacement**. Detecting and replacing known objects with parametrised representations could be done. (Similar to above.) Such parametrised objects might include extra details, details that do not actually exist, animations, interactivity or other features to improve a virtual representation. This could potentially cover non-solid continuous objects, such as water.

- **Safe Viewing**. Most models of real world data will have gaps. Letting users see the gaps in the model would be unacceptable; a means of calculating the safe areas for a user to be in would be required. In addition this would provide a diagnostic, for finding gaps in the model so data can be captured to fill them. This could also cover detail, so the user can not get too close to low detail objects. The automatic generation of a sky sphere[3] to fill in the background would be required for completeness.

- **Simplification**. Once a 3D model is produced simplification will be required in most cases, especially for real-time rendering purposes. Whilst simplifying 3D models is a well understood area the metrics used are general, for a navigable virtual environment different metrics might be optimal. For instance, if an aviator is to collide with the model then removing certain smaller concave shapes would be advantageous to avoid the aviator getting stuck in/on them.

- **Non-visual Material Properties**. In addition to detecting the visual properties of materials other properties exist that could be determined using pattern recognition. For instance, detecting that a surface is wood allows for a realistic footstep sound when walking on it within a virtual environment. Such information could also assist in deciding which objects should be solid to the aviator, so the floor should be solid but bushes are usually not, as you can get stuck on there irregular shape.

[3] A rendering of objects at a distance where perspective is no longer an issue, rendered as an unmoving sphere around the user.

Beyond the improvements mentioned so far if rewrites of large parts of the system were an option then the following could be considered:

- **Interface**. The current system is suitable only for people who can use XML and have an understanding of how the system works. Putting an interface on top of such a system could open it to a larger number of users. This could be done by adding extra hooks into the framework to keep the interface as a separate module.
- **Human Interaction**. As stated in subsection 3.7.1 this would be advantageous to correct the mistakes of the algorithms.
- **Distributed Computing**. In addition to swapping in faster algorithms bringing more CPU cycles to task would be advantageous. By changing the framework appropriately but maintaining the interface this could be done without rewriting any of the algorithms.

Bibliography

[1] Direct dimensions (company). http://www.directdimensions.com/prod_scan.htm. Last accessed: 13th October 2004.

[2] Eyetronics. Shapesnatcher (product). http://www.eyetronics.com/products/shapesnatcher.php. Last accessed: 13th October 2004.

[3] 3D Scanners. Modelmaker (product). http://www.3dscanners.com/. Last accessed: 13th October 2004.

[4] Direct Dimensions. Direct dimensions 2004 product catalog. http://www.directdimensions.com/pdfs/ProductCat_W2004.pdf. Last accessed: 13th October 2004.

[5] Marc Pollefeys. Visual 3d modeling from images. http://www.cs.unc.edu/%7Emarc/tutorial.pdf. Last accessed: 13th October 2004.

[6] Richard Szeliski Yanghai Tsin, Sing Bing Kang. Stereo matching with reflections and translucency. *IEEE Conference on Computer Vision & Pattern Recognition*, pages 702–709, 2003.

[7] Zhengyou Zhang. A flexible new technique for camera calibration. http://research.microsoft.com/~zhang/Papers/TR98-71.pdf (Last Accessed: 2nd February 2005).

[8] Q.-T. Luong O.D. Faugeras and S.J. Maybank. Camera self-calibration: Theory and experiments. *Lecture Notes in Computer Science, Vol. 558; European Conference on Computer Vision*, pages 321–334, 1992.

[9] A high resolution 3d surface construction algorithm. http://www.cs.technion.ac.il/~u_shani/cs236807-S2/lectures-students/Marching%20Cubes.pdf. Last accessed: 13th February 2005.

[10] Chris Harris & Mike Stephens. A combined corner and edge detector. *Proceedings of The Fourth Alvey Vision Conference*, pages 147–151, 1988. http://www.csse.uwa.edu.au/~pk/Research/MatlabFns/Spatial/Docs/Harris (Last Accessed: 9th February 2005).

[11] J. P. Lewis. Fast normalized cross-correlation. 1995. http://www.idiom.com/~zilla/Work/nvisionInterface/nip.html (Last Accessed: 9th February 2005).

[12] Stan Birchfield. An introduction to projective geometry (for computer vision). http://robotics.stanford.edu/~birch/projective/projective.pdf. Last accessed: 20th January 2005.

[13] Ethan Gold. Building a homemade laser line scanner. http://www.thaumaturgy.net/~etgold/scanner/. Last accessed: 10th February 2005.

[14] Faro platinum arm (product). http://www.directdimensions.com/pdfs/DDI%20-%20FARO%20Platinum%20Arm.pdf. Last accessed: 10th February 2005.

[15] Qinetiq. Passive millimetre wave systems (technology). http://www.qinetiq.com/home/core_skills/sensors_and_electronics/optronics/mmw_systems.html. Last accessed: 10th Feburary 2005.

[16] Ramani Duraiswami. Epipolar geometry and the fundamental matrix. http://www.umiacs.umd.edu/~ramani/cmsc828d/lecture27.pdf. Last accessed: 24th January 2005.

[17] Roger Boyle Milan Sonka, Vaclav Hlavac. *Image Processing, Analysis, and Machine Vision*. PWS Publishing, 1999.

[18] *Numerical Recipes in C: The Art of Scientific Computing*, chapter 2.6. Cambridge University Press, 1988. http://www.library.cornell.edu/nr/bookcpdf/c2-6.pdf Last accessed: 25th February 2005.

[19] Daniel DeMenthon. Reconstruction from multiple views. http://www.umiacs.umd.edu/~ramani/cmsc828d/lecture28.pdf. Last accessed: 24th January 2005.

[20] Richard Wilson. http://www-course.cs.york.ac.uk/cvi/slides/cvi3.pdf. Last accessed: 25th February 2005.

[21] Jana Kosecka. Two-view geometry. http://cs.gmu.edu/~kosecka/lect4.ppt. Last accessed: 15th March 2005.

[22] Paulo R. S. Mendonca Kwan-Yee K. Wong and Roberto Cipolla. Camera calibration from symmetry. ftp://swr-ftp.eng.cam.ac.uk/pub/reports/wong_ima00.pdf (Last Accessed: 11th February 2005).

[23] Christian Perwass Bodo Rosenhahn and Gerald Sommer. Foundations about 2d-3d pose estimation. http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/ROSENHAHN1/CVOnlinePose.html (Last Accessed: 26st February 2005).

[24] John Ens and Peter Lawrence. An investigation of methods for determining depth from focus. *IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 15, No. 2*, pages 97–107, 1993.

[25] P.L.Worthington and E.R. Hancock. Coarse view synthesis using shape-from-shading. *Pattern Recognition 36*, pages 439–449, 2003. http://www-users.cs.york.ac.uk/~erh/rae/cvs.pdf (Last Accessed: 17th March 2005).

[26] Reinhard Koch Marc Pollefeys and Luc Van Gool. A simple and efficient rectification method for general motion. *7th International Conference on Computer Vision*, pages 496–501, 1999.

[27] Adam Baumberg. Reliable feature matching across widely separated views. *IEEE Proceedings on Computer Vision & Pattern Recognition Vol. 1*, pages 774–781, 2000.

[28] S. J. Cunnington and A. J. Stoddart. N-view point set registration: A comparison. http://www.ee.surrey.ac.uk/Research/VSSP/3DVision/virtuous/Publications/cunnington-bmvc99.pdf (Last Accessed: 21st February 2005).

[29] Richard Szeliski Heung-Yeung Shum Lifeng Wang, Sing Bing Kang. Optimal texture map reconstruction from multiple views. *IEEE Conference on Computer Vision & Pattern Recognition, Vol. 1*, pages 347–354, 2001.

[30] Peter Schroder Igor Guskov, Wim Sweldens. Multiresolution signal processing for meshes. *Computer Graphics Proceedings, Annual Conference Series*, pages 325–334, 1999.

[31] Jitendra Malik Paul E. Debevec, Camillo J. Taylor. Modeling and rendering architecture from photographs: A hybrid geometry- and image-based approach. *Computer Graphics Proceedings, Annual Conference Series*, pages 11–20, 1996.

[32] Ian Sommerville. *Software Engineering*. Addison Wesley, 2001.

[33] Worldwide Web Consortium. Extensible markup language. http://www.w3.org/XML/. Last accessed: 2nd March 2005.

[34] Worldwide Web Consortium. Document object model. http://www.w3.org/DOM/. Last accessed: 2nd March 2005.

[35] Daniel Scharstein and Richard Szeliski. http://bj.middlebury.edu/~schar/stereo/web/results.php. Last accessed: 3rd March 2005.

[36] George Chen Li Hong. Segment-based stereo matching using graph cuts. *IEEE Conference on Computer Vision & Pattern Recognition, Vol. 1*, pages 74–81, 2004.

[37] Satish B. Rao Ingemar J. Cox, Sunita L. Hingorani and Bruce M. Maggs. A maximum likelihood stereo algorithm. *Computer Vision and Image Understanding, Vol. 63, No. 3*, pages 542–567, 1996.

[38] Lutz Falkenhagen. Hierarchical block-based disparity estimation considering neighbourhood constraints. *IEEE Workshop On Multimedia Signal Processing*, 1997.

[39] Lutz Falkenhagen. Depth estimation from stereoscopic image pairs assuming piecewise continuos surfaces. *Image Processing for Broadcast and Video Production*, pages 115–127, 1994.

[40] Robert D. Dony Wenxin Wang. Evaluation of image corner detectors for hardware implementation. http://www.soe.uoguelph.ca/webfiles/rdony/pubs/WangCCECE04.pdf (Last Accessed: 13th December 2004).

[41] Robert B. Fisher. The ransac (random sample consensus) algorithm. http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/FISHER/RANSAC/. Last accessed: 5nd March 2005.

[42] Nicholas Ayache and Charles Hansen. Rectification of images for binocular and trinocular stereovision. *9th International Conference on Pattern Recognition*, pages 11–16, 1988.

[43] Ingemar J. Cox Sebastien Roy, Jean Meunier. Cylindrical rectification to minimize epipolar distortion. *IEEE Conference on Computer Vision & Pattern Recognition*, pages 393–399, 1997.

[44] David Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, pages 91–110, 2004. http://www.cs.ubc.ca/~lowe/papers/ijcv04-abs.html (Last Accessed: 15th March 2005).

[45] Mark Priestley. *Practical Object-Oriented Design with UML*. Mc Graw Hill, 1996.

Appendices

A Aegle[1] Structure

A.1 Overview

This section follows on from section 4.3 by detailing exactly how the framework is implemented. It first covers cross cutting issues such as the types and data structures used throughout, then it follows the design structure, detailing the DOM, the single variable type and the module interface in turn. Finally it lists the standardised variable types and how to implement new modules for the system.

UML[45] has been used to provide a diagrammatic representation of the systems under discussion due to it being a well known notation. (This is not to imply that UML has been used as a serious design tool, the system was not complicated enough to warrant that level of formality.) I have assumed knowledge of `C` and theoretical Object Oriented Programming for this section. (`C++` would be of greater use.) Attempts have been made to explain more obscure techniques when they arise.

A.2 Cross-Cutting Concerns

Any system has a set of cross-cutting concerns in its implementation, detailed here as a series of bullet points:

- All classes are preceded with AE to make sure there names are unlikely to clash with any other system it is integrated with. `namespaces` were not used as a matter of preference.
- A consequence of using dynamically linked code under Windows^{tm} is that any memory declared in a module must be deleted by that module. To facilitate this two functions `AEmalloc` and `AEfree` plus a base class `AEbase` (See Figure A.1) are exported from the executable with which all modules statically link. All memory is then allocated and freed within the executables heap, circumventing the problem.

[1] The codename given to the framework, as a consequence all classes etc. in the system are given the prefix `AE`. The source is from Greek mythology, where it is a name shared by several gods. In this particular instance it references one of the Heliades, a daughter of Helios the Sun God. It means 'light, radiance, glory'.

- Instead of using C++ types directly they have all been redefined, to ensure handling different compilers, operating systems and 64 bit processors is possible. These types are as follows, there meaning is presumed obvious or explained in other parts of this section:

```
AEbool   AEuint16  AEuint64  AEstring
AEint8   AEint32   AEreal32  AEconstString
AEuint8  AEuint32  AEreal64  AEbyte
AEint16  AEint64   AEchar    AEtoken
```

- The DOM among other parts has to handle strings, this is always problematic as strings are of variable size, slow to compare and hard to manage. A solution to this is to hash all strings to numbers, then to manage them instead. To this end `AEtokenTable` (Figure A.1) and a `typedef` of `AEtoken` to an integer is provided. For when strings have to be used a `typedef` of `AEstring` to `char*` is provided. `AEconstString` to `char* const` is also provided. The token table class is a `functor`[2] As a function it converts a string into a unique number (A `AEtoken`), it also provides the method `Str` to do the inverse.

- Hash tables have been used extensively throughout the system. These are provided by the `AEdenseHash*` series of flat[3] templates[4]. There are four of these templates, each one providing a different kind of deletion of the objects it contains:

`AEdenseHashPtr`	Does not delete its contents on destruction.
`AEdenseHashFree`	Uses `AEfree`.
`AEdenseHashDel`	Uses `delete`.
`AEdenseHashDelArray`	Uses `delete[]`.

  They hash from numbers to the contained type, see Figure A.1 for the interface. Its main feature is `operator[]` access to references to pointers of its type, so it behaves as a sparse array. Due to it returning references entries can be set to null, requiring the provision of methods to clean out unused indexing data.

[2] A functor is a C++ specific way of implementing a function with state. By way of example in this case given a `'AEtokenTable t'` you can call `'t("wibble")'` to get a unique number for the given string.

[3] A flat template is a template with no code. Essentially a normal class is implemented such that it will work with everything (This is inevitably nasty.), then a template class inherits from it and provides a clean interface with every method inline.

[4] Generics in C++

Figure A.1: UML Class Diagram of Basic Types

A.3 The Document Object Model

The DOM (http://www.w3.org/DOM/) is a standard construct but for the purposes of this project a complete one is not required whilst some variations are. It should be instantly recognisable to anyone familiar with this area however. See Figure A.2 for the UML diagram. Two abstract classes, `AEitem` and `AEtree`, exist to represent lists of items and trees of items respectively. `AEattrib` represents an attribute within an element, and *is a* `AEitem` as each element contains a list of them. `AEelem` represents an element, and as elements form a tree *is a* `AEtree`. `AEitem` and `AEtree` provide the standard linked list and tree browsing manipulation methods.

`AEattrib` provides the `Name` method which returns a token of its name and a set of methods to get the data associated with that name. These include string access as normal but also includes other methods to convert to integers/reals etc.. These methods cache there result, so the conversion is only done the first time the method is called.

`AEelem` has three sets of methods. The first set is inherited from `AEtree`, the second set mirror the `AEattrib` data getters, except they take an attribute name

token for which attribute of that element to access and a default value for if the attribute does not exist. The final pair of methods allow accessing of sub-elements of the element by name, with indexing for if multiple sub-elements exist with the same name. To load the XML file two functions are also provided:

```
AEelem * ParseXML(AEstring xml,AEtokenTable & tokTab);
AEelem * LoadXML(AEstring filename,AEtokenTable & tokTab);
```

They construct a DOM from there input, respectively a null terminated string and a file name to be loaded and parsed.

A.4 The Single Variable Type

The SVT is a hierarchy of nodes, with leaf nodes that contain data. It is made of three classes, `AEnode` which represents any node in the hierarchy and then the two types of node that can exist, `AEbranch` and `AEleaf`. See Figure A.3 for the UML. As the SVT is constructed/de-constructed by modules `AEnode` inherits from `AEbase`.

`AEnode` is `virtual` (Abstract in UML terminology.) and provides methods to indicate if its a branch or a leaf. As both branches and leaves have types it provides methods for getting and setting type, it also provides a recursive `Clone()` method for duplicating parts of a hierarchy. `AEbranch` and `AEleaf` inherit from this and implement there own versions of these methods.

`AEbranch` contains a list of `AEnodes`. This list can be numerically accessed using `operator[]` and edited using `Add(...)`,`Del(...)` and `Remove(...)` methods. (`Del(...)` removes the `AEnode` from the data structure and then deletes it, `Remove(...)` only removes it so it can be `Add(...)`ed again.) It also provides methods (`Del(...)` & `Get(...)`) to index by type. In addition `Read(...)` and `Write(...)` methods are provided so the SVT object can be saved and loaded to/from disk. (A constructor is also provided to construct the object from disk.)

One of the properties of `AEleaf` is that once constructed its data can be changed but its meta-data can not. Due to the quantity of meta-data associated with each `AEleaf` a support class, `AEleafDesc` is provided to be passed into the constructor of `AEleaf` and define its structure. It provides the following public variables:

`.type`	The node type, see section A.6.
`.dims`	How many dimensions it has. A list would be one, an image two and a voxel field three for instance.
`.size[]`	This is a pointer to an array of `AEint32`, each element being the size of that dimension. (The array must contain .dims valid entries.) On construction it points to an array of three AEint32's contained in this structure. In the event that more dimensions are needed it must be set to point to a different array, of which it is the class users responsibility to free.
`.fields[]`	An array of booleans indexed by field type. For the structure to include a particular field set the relevant value to true, otherwise leave it as false.

On construction a `AEleafDesc` has the entire `.fields[]` array set to `false`, so you only have to set to `true` the fields required.

For `AEleaf` three constructors are provided, the primary one that uses `AEleafDesc` a copy constructor and a constructor to load in a saved `AEleaf`. A `Save(...)` method is also provided to store the data in the first place. (`Load(...)` does not make sense, as a `AEleafs` structure can not be modified after construction.) A set of methods are provided to then get the `AEleaf` structure, including extracting a `AEleafDesc` object so it can be edited and used to construct similar `AEleafs`. The data area provided allows overrun up to a certain limit, changed by `MinBorder(...)`. The postcondition of calling this is that you can be the given value outside the limits of the data without causing problems. The `Get*(...)` methods form the bulk of the interface and all return pointers to data, so you can edit the data as well as read it. Each of the methods in this set has two properties, of which every combination is provided; the first property is the dimensionality of the access - there are special once for 1D, 2D and 3D to cover the majority of accesses, there is also *n*D accessors for if more dimensions are required or an algorithm that works within multiple dimension counts is implemented. The second property is the return type of the pointer, which can be `void*`, `AEbyte*`, `AEint32*`, `AEuint32*` or `AEreal32*`. In addition to the standard methods there are also a set of specialist methods, for very particular situations, all of which were implemented as needed:

`.Grab(...)`	Given another `AEleaf` with the same dimensionality this copies all shared fields from that `AEleaf` into this one.
`.Set(...)`	Given a field and some data this sets every instance of that field in the structure to that data.
`.MallocField(...)`	This is given a specific field, it then mallocs a tightly packed array of all the data in that field from the structure and returns it.
`.Fetch(...)`	This performs similarly to `.Grab(...)`, except it only works on a single passed in field.
`.Line(...)`	Limited to 2D data sets only, sets the value of a passed field along a line to passed data. Exists specifically for drawing lines on images.

In addition to the interface provided by `AEleaf` a further support class exists, `AEiter`. This iterates all the entries in the `AEleaf` it is constructed with. It is designed to be constructed and then used in the form

```
do {
...
} while (iter.Next());
```

Where `.Next()` moves to the next item and returns true, unless its the last item in which case it returns to the start of the data and returns false. It provides similar `.Get*(...)` methods to `AEleaf` except without coordinates being required. It also provides offset Gets in 1D and 2D, to get at the item above or to the left for instance.

A.5 The Core

`AEcore`, see Figure A.4, unifies all the aforementioned classes to create the system as a whole. A single one is constructed within the executable and provides all services to loaded in modules and executes the script. Its parts are as follows:

- **Token Hashing**. The `AEcore` class inherits from `AEtokenTable`, so that it provides all the token services that various parts of the system require.
- **Module loading**. This consists of three methods. `AddOp(...)` registers an operation with the system, giving it the relevant method and name with which its referenced. The core maintains a list of all registered operations so when asked to execute a XML chunk it can. The second method is `LoadModule(...)`, this is given the name of a dynamically linked library, it then links in that library and calls its handle function, which then calls the `AddOp(...)` method to register all the operations it contains. `LoadModules()` looks in the working directory, finds all dynamically linked libraries and calls `LoadModule(...)` on them.

- **Execution**. Two methods are provided to execute the scripting language, `ExecOp(...)` is given a DOM element, it expects a `<op ...>` element and manages scope automatically. `Exec(...)` is given a DOM element, the details of which do not matter, it loops through all sub-elements in order and executes each in turn, including handling `<var ...>` elements. Both of these methods return `true` on error, indicating either an unrecognised element, operation or an operation indicating that an error happened.

- **Scope**. The scripting language structure requires scoping of variables, this is provided by two methods, `PushScope()` and `PopScope()`. To manage which variables go into and out of the current scope the method `Link()` explicitly generates a variable in the current scope that aliases a variable in the previous scope.

- **Variables**. SVT's can be created and named using the `NewVar(...)` method, once created they can be obtained by name using `Var(...)`, noting that you can only access variables either created in the current scope or linked from a previous scope. A newly created SVT is a empty `AEbranch` with its type set to `Tnone`.

- **Operation Parameters**. To assist with passing parameters into and out of operations two sets of arrays are provided, an array of in variables and an array of out parameters, both of which are pushed/popped by the stack, the methods `Set/Get In/Out Param(...)` are the interface for this functionality.

- **Logging**. A single method, `Msg(...)` is provided that outputs any text given to it so the user can see it. It uses the `printf` interface.

Using this interface executing an Aegle script consists of loading the relevant script into a DOM, creating a `AEcore` and calling its `LoadMethods()` method, then finally calling `ExecOp(...)` on the root element of the loaded DOM.

A.6 Variable Types

The SVT by its nature allows for arbitrary data structures. Passing data between operations however requires standard data structures to be defined. To this end a standard list of fields is defined and a standard list of SVT types that covers all arrangements of data passed around. This is also advantageous for error checking, as any structure of a particular type can have certain assumption made about its structure instead of every operation having to check all details. Lists for both follow:

A.6.1 Fields

`Fbyte`	A simple `AEbyte`, for storing arbitrary data.
`Fflag`	A set of 8 flags in a `AEbyte`, for masks and the like.
`Fcount`	A `AEuint32` counter, for histograms.
`Fvalue`	An arbitrary `AEreal32`, main use as components of a matrix.
`Findex1`	An index to a 1 dimension/position structure. 1*`AEuint32`. (Used for texture indices.)
`Findex2`	An index to a 2 dimension/position structure. 2*`AEuint32`. (Used for edges indices.)
`Findex3`	An index to a 3 dimension/position structure. 3*`AEuint32`. (Used for indexed triangles.)
`Findex4`	An index to a 4 dimension/position structure. 4*`AEuint32`. (Used for indexed quads)
`FcolourL`	Luminescence component, `AEbyte`, for greyscale images.
`FcolourRGB`	3 `AEbytes`, RGB.
`FcolourA`	An alpha/transparency component, to compliment the colour spaces, `AEbyte`.
`Flength`	A `AEreal32` length.
`FposX,FposY,FposZ,FposW`	The components of a X, Y, Z, W position vector, `AEreal32`s.
`FdirX,FdirY,FdirZ,FdirW`	The components of a X, Y, Z, W direction vector. Used for normals among other purposes, `AEreal32`s.
`FtexS,FtexT,FtexU`	Components of a S, T, U texture coordinate, `AEreal32`s.
`FangH,FangP,FangB`	Components of a H, P, B angle using *AEreal32*s. (Anticlockwise, radians)

A.6.2 Types

Basic Types

- `Tnone`. Used on construction of nodes, no other purpose.
- `Tarray`. An array of other types. (`AEbranch` only)

Leaf Types

- `Tmatrix`. A matrix, must be 2D and only contain field `Fvalue`.
- `TimageL`. A greyscale image. Is required to be 2D, with `FcolourL` set. Optionally `FcolourA` can be included if an alpha map is required, `Flength` can be included for a depth map and `Fflag` can indicate a mask, with any bits set indicating membership.

- `TimageRGB`. Similar to `TimageL` except its for colour images, only difference is it requires `FcolouRGB` instead of `FcolourA`. The same optional fields apply.
- `Tdepth`. A depth map, consists only of the `Flength` field in a 2D structure. No optional fields.
- `Tstencil`. A mask, a set of `Fflag` fields in a 2D arrangement. Having any bits set in `Fflag` indicates membership.
- `ToffsetImage`. A 2D structure containing fields `FposX` and `FposY`. This is the output of a stereo correspondence algorithm and maps each pixel in the structure to its correspondence, indexed by the fields. Pixels with no corresponding pixel have both there fields set to negative numbers.
- `TvoxelL`. A 3D `TimageL`.
- `TvoxelRGB`. A 3D `TimageRGB`.
- `Tvolume`. A 3D `Tmask`.
- `Thistogram`. A 1D set of `Fcount`, representing the sum of instances of a range of cases.
- `TshapedTexture`. An image (2D) with 3D coordinates associated with each pixel as well as colour. Requires a 3D coordinate and colour, can also have a `Fflag` to indicate coordinate validity. Specifically created to supersede depth maps, as latter iterations of the code generate 3D coordinates directly from correspondences.

3D model

A 3D model consists of a set of leafs inside a branch with type `Tmodel3D`. In addition to the below listed leaf types images can also be included as textures. A 3D model consists of a set of textures, vertices, edges (Line segments), triangles and quads. Any image or voxel type included in the branch is a texture to be referenced by the geometry, indexed with 0 being the first one within the branch.

- `Tpoints3D`. This is a 1D array of vertices. It requires a position consisting of `FdirX`, `FdirY` and `FdirZ`, optionally `FdirW`. It can also contain the `Fdir` set as a normal for that vertex, the `Ftex` set for texture coordinates and any of the three colour field types for per vertex colour/transparency. These vertices are then indexed by other parts of the 3D model, the indices start at 0 with multiple `Tpoints3D` being appended together to create the complete set in the order they are stored in the branch. The next type, `TpointsExtra3D`, is also included in that array.

- `TpointsExtra3D`. The extra points type also adds vertices, it however defines them in terms of vertices already defined via the required `Findex1` field it has, overriding them with the same optional fields as `Tpoints3D`. The reason for this is it allows you to define one vertex with multiple normals/texture coordinates/colours assigned to it such that a renderer knows they are the same vertex and can optimise accordingly.

- `Tedges3D`. This is a 1D array of `Findex2` that indexes vertices to define the start and end of each edge. It can also have a colour associated with it to override anything set for the vertices. Edges can not be textured.

- `Ttris3D`. This defines a 1D list of primitives, each with a `Findex3` to define its three corners. It can also have a `Findex1` to indicate it should be textured with the texture of that index, colour information to colour it and the `Fdir` set to flat shade it with a given normal.

- `Tquads3D`. Identical to `Ttris3D` except it uses the `Findex4` field instead to indicate four vertices.

A 3D model can be explicitly rendered as either a point field, a set of edges or a set of primitives.

A.7 Module Implementation

To add an operation to Aegle involves creating a module with that operation (and usually others) in it. An operation is a function with the following prototype:

```
AEGLE_FUNC AEbool MyOperation(AEcore & core,AEelem & self);
```

When an operation is called it is given the core object as its interface to the system, and self, the DOM element that caused this operation to be called. It should return `false` if everything is ok or `true` if something goes wrong[5]. Operations usually all take the same sequence of steps:

- Read and validate the variables passed in to and out of the operation. (in="" and out="" in the element self, using the `Get/Set In/Out Param()` core methods.)

- Uses `self` to get specific parameters to control the execution of the operation.

- Loops all data passed in via the in SVTs and processes it, adding output data to the out SVTs.

[5]Standard C++ `throw/catch` were avoided as I have no experience of using them within dynamically linked libraries, and could not risk the potential time lost if it proved problematic.

- Returns `false` on success, or uses `core.Msg` to give a reason for failure and returns `true`.

The operations need to be compiled into a dll, with a handle function:

```
AEGLE_FUNC void OnAegleLoad(AEcore & core)
{
  core.AddOp(core("MyModule.MyOperation"),MyOperation);
  ...
}
```

Aegle will then automatically load the dll if its put into the working directory and you can use the operation in a script file with the XML fragment

```
<op name="MyModule.MyOperation" in="..." out="..." .../>
```

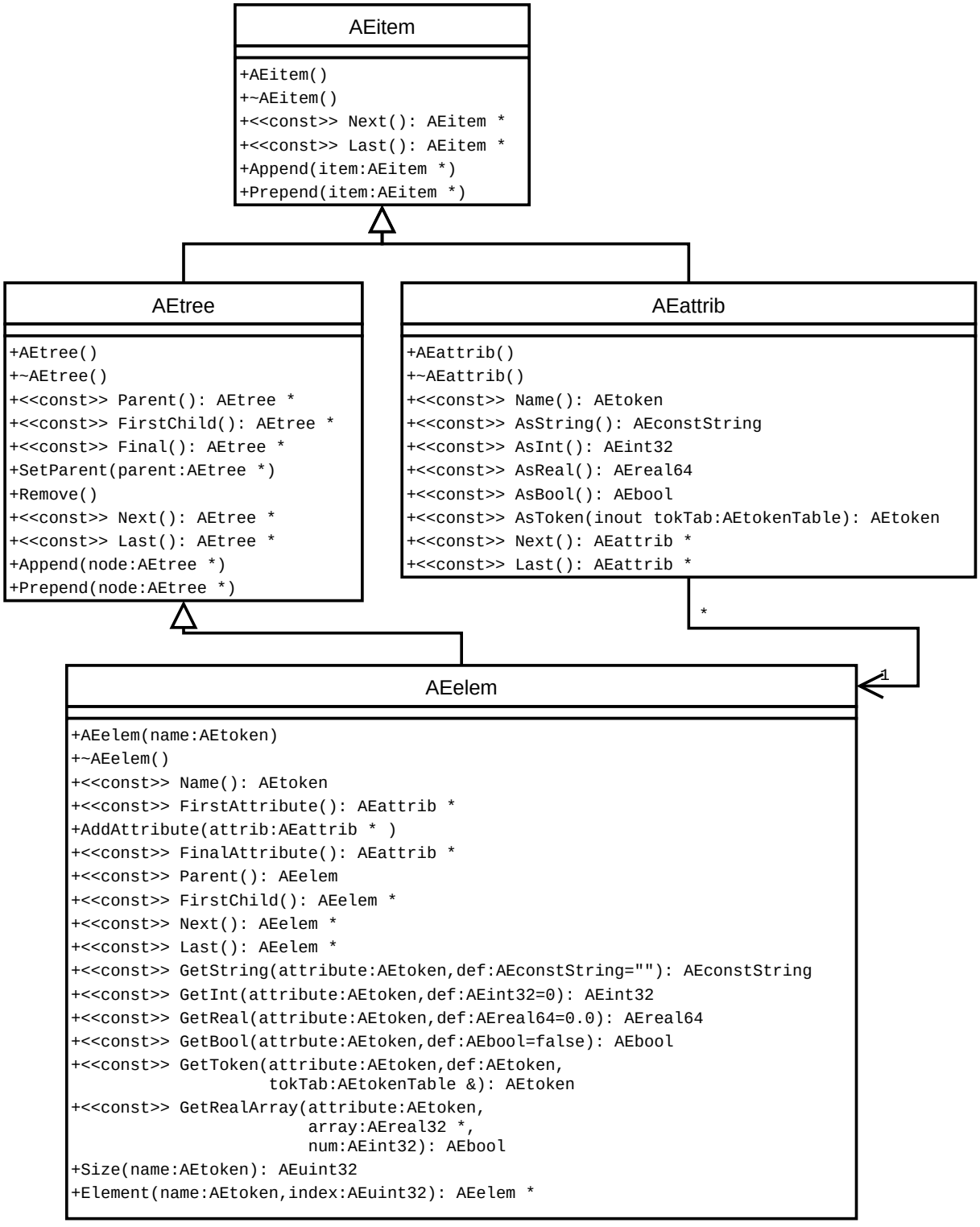


Figure A.2: UML Class Diagram of the DOM

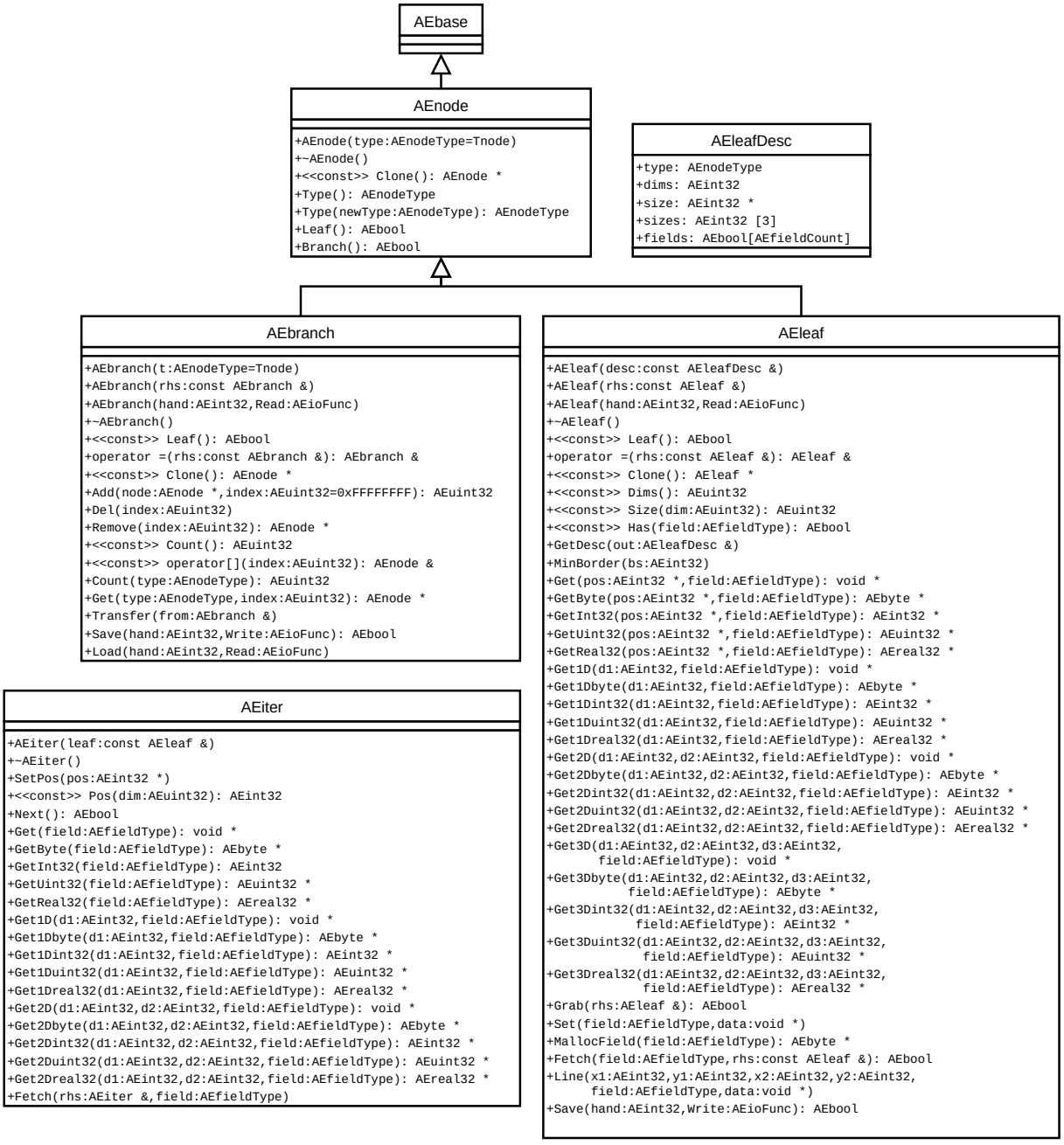


Figure A.3: UML Class Diagram of the SVT

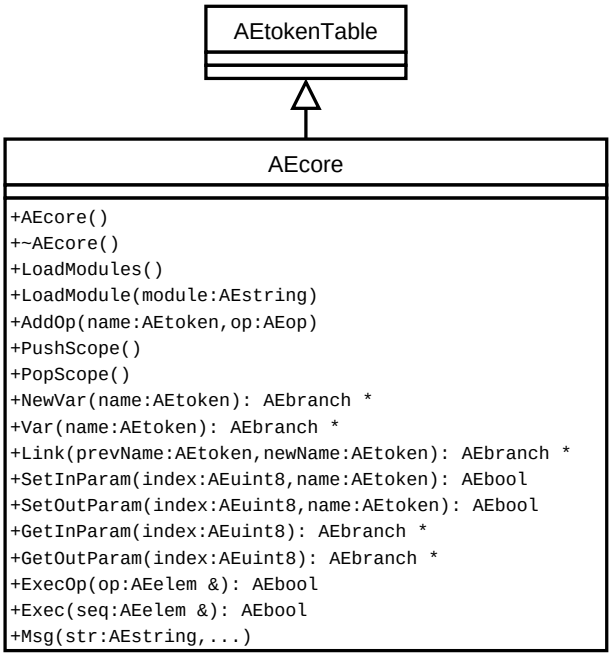


Figure A.4: UML Class Diagram of the Core

B Aegle Operations

The following chapter details all operations that have been implemented, subdivided by package.

B.1 Embedded

These three operations are integrated into the Aegle executable as there basic functionality.

exec:

```
<op name="exec" uses="...">
    <!- Variables & Operations ->
</op>
```

This executes the operations contained within inside a new scope.

uses: This contains a comma separated list of variables to import into the scope, as external variables are not implicitly available.

time:

This is identical to `exec`, but it also times how long the operations within take and prints it out to the console.

msg:

```
<op name="msg" msg="Hello World!"/>
```

Prints a message to the console.

msg: The message to be printed.

B.2 var

This module contains general purpose operations, that work on all variables.

var.copy:

```
<op name="var.copy" in="from" out="to"/>
```

Copies a variable into another variable, overwriting the contents of the `to` variable.

in[0]: The variable to be copied.
out[0]: The variable to be overwritten.

var.append:

```
<op name="var.append" in="from" out="to"/>
```

Clones the root nodes children of `from` and appends them to the end of the root node of `to`.

in[0]: The variable to be copied.
out[0]: The variable to be merged into.

var.save:

```
<op name="var.save" in="var" filename="file"/>
```

Serialises a variable into a file so it can be latter loaded.

in[0]: The SVT to save.
filename: The file the SVT will be saved into.

var.load:

```
<op name="var.load" out="var" filename="file"/>
```

Loads a file putting the SVT it contains into the given variable.

out[0]: The variable to load into.
filename: The file the SVT will be loaded from.

B.3 image.io

This module provides two methods, one to load images and another to save images. Uses Developers Image Library[1] and will load/save most image formats. When saving it decides the type based on the extension.

[1] http://openil.sourceforge.net/

image.io.load:

```
<op name="image.io.load" out="loaded">
    <file value="image.jpg"/> <!- Repeatable ->
</op>
```

Loads image files, is given a list of images which it loads into the root of the out variable. The images loaded will be colour or greyscale depending on the file format, and will have alpha channels if the file does.

out[0]:	The variable to load into.
file:	Each file element loads a file, the name given by the value attribute.

image.io.save:

```
<op name="image.io.save" in="images">
    <file value="image.jpg" index="0"/> <!- Repeatable ->
</op>
```

Saves given images, as the given data can contain multiple image leafs this operation is designed to be able to save multiple files.

in[0]:	The variable to save from.
file:	Each file element saves a file, the name given by the value attribute and the index giving which file in the set to save.

B.4 math.matrix

This section provides some matrix manipulation operations.

math.matrix.identity:

```
<op name="math.matrix.identity" out="matrix" width="4"
height="4"/>
```

Creates an identity matrix.

out[0]:	The output matrix.
width:	The width of the output matrix.
height:	The height of the output matrix.

math.matrix.transform:

```
<op name="math.matrix.transform" in="in" out="out">
    <rotate h="180" p="90" b="0"/>
    <offset x="10" y="5" z="0"/>
    <scale x="2" y="2" z="2"/>
</op>
```

This applies transforms to a 4x4 matrix, in the order given.

in[0]:	(Optional) Matrix to transform. If omitted it starts from an identity matrix.
out[0]:	Output matrix.
<*rotate*>:	Does a HPB rotation.
<*offset*>:	Does a translation.
<*scale*>:	Does a scaling.

math.matrix.add:

```
<op name="math.matrix.add" in="in1,in2" out="out"/>
```

Calculates out = in1 + in2.

in[0]:	First matrix.
in[1]:	Second matrix.
out[0]:	Resultant matrix.

math.matrix.sub:

```
<op name="math.matrix.sub" in="in1,in2" out="out"/>
```

Calculates out = in1 - in2.

in[0]:	First matrix.
in[1]:	Second matrix.
out[0]:	Resultant matrix.

math.matrix.mult:

```
<op name="math.matrix.mult" in="in1,in2" out="out"/>
```

Calculates out = in1 * in2.

in[0]:	First matrix.
in[1]:	Second matrix.
out[0]:	Resultant matrix.

math.matrix.transpose:

```
<op name="math.matrix.transpose" in="in" out="out"/>
```

Calculates the transpose of a matrix.

in[0]:	Input matrix.
out[0]:	Output matrix.

math.matrix.inverse:

```
<op name="math.matrix.inverse" in="in" out="out"/>
```

Calculates the inverse of a matrix.

in[0]:	Input matrix.
out[0]:	Output matrix.

math.matrix.shrink:

```
<op name="math.matrix.shrink" in="in" out="out" width="5"
height="3"/>
```

Shrinks a matrix to make it smaller, useful for trimming data matrices.

in[0]:	Input matrix.
out[0]:	Output matrix.
width:	(Optional) Width to set the matrix to, will not make it larger and if omitted its width is not changed.
height:	(Optional) Height to set the matrix to, will not make it larger and if omitted its height is not changed.

math.matrix.save:

```
<op name="math.matrix.save" in="matrix" filename="out.csv"/>
```

Saves a matrix to a comma separated file.

in[0]:	Input matrix.
filename:	File matrix is saved to.

math.matrix.load:

```
<op name="math.matrix.load" out="matrix" filename="out.csv"/>
```

Loads a matrix from a comma separated file.

out[0]:	Output matrix.
filename:	File matrix is loaded from.

B.5 stats

A set of operations to determine the covariance matrices required by certain versions of the stereo algorithms.

stats.covariance.imageRGB:

```
<op name="stats.covariance.imageRGB" in="image1,image2,..."
out="matrix"/>
```

Takes a selection of RGB images as inputs and calculates a covariance matrix on the 3 colour components, making a 3x3 matrix.

in[n]: Input images to be sampled.
out[0]: Resulting 3x3 matrix.

stats.covariance.imageRGB.diff:

```
<op name="stats.covariance.imageRGB.diff" in="image1,image2"
out="matrix"/>
```

Calculates the covariance of the differences between two RGB images.

in[0]: Input colour image 1.
in[1]: Input colour image 2.
out[0]: Resulting 3x3 matrix.

stats.covariance.imageArea33.diff:

```
<op name="stats.covariance.imageArea33.diff" in="image1,image2"
out="matrix"/>
```

Calculates the covariance of the differences between 3x3 areas of the given greyscale images.

in[0]: Input greyscale image 1.
in[1]: Input greyscale image 2.
out[0]: Resulting 9x9 matrix.

B.6 image

Some miscalaneous image operations.

image.set.depth:

```
<op name="image.set.depth" in="image,depth" out="merged"/>
```

Takes an image and a depth map and merges them to produce an image with depth

in[0]: Input image map, either colour or greyscale.
in[1]: Input depth map. Must be the same size as the image map.

out[0]: Resulting `Timage*` width depth data..

image.from.depth:

```
<op name="image.from.depth" in="depth" out="image"/>
```

Generates an image from a depth map, sets nearby objects to blue then fades to red as they get further away and then to green as they get further still.

in[0]: Input depth map.
out[0]: Resulting image.

B.7 image.filter

Filters for images, for filtering them prior to other processing, such as Gaussian blurs before correspondence matching.

image.filter.grey:

```
<op name="image.filter.grey" in="colour" out="grey"/>
```

Converts a colour image into a greyscale image. If its already greyscale it just copies it.

in[0]: Input image.
out[0]: Output image.

image.filter.colour:

```
<op name="image.filter.colour" in="grey" out="colour"/>
```

Converts a greyscale image into a colour image. If its already colour it just copies it.

in[0]: Input image.
out[0]: Output image.

image.filter.convolve:

```
<op name="image.filter.convolve" in="in" out="out">
    <kernel width="3" height="3" data="0,1,0,1,2,1,0,1,0"
scale="1.0"/>
</op>
```

This applies a sequence of convolutions to the input. The kernels are normalised

and then multiplied by the given scale. Works on all image channels available, including alpha.

in[0]:	Input image.
out[0]:	Output image.
<kernel>:	Multiple kernels can listed, each has a window (Sizes must be odd) defined and factors for that window, plus the scale value.

image.filter.gaussian:

```
<op name="image.filter.gaussian" in="in" out="out" sd="3.2"/>
```

Applies a Gaussian blur to the input image.

in[0]:	Input image.
out[0]:	Output image.
sd:	The standard deviation of the blue, in pixels.

image.filter.threshold:

```
<op name="image.filter.threshold" in="in" out="out" min="0"
max="255"/>
```

Not really a threshold unless min==max, sets any pixel less than or equal to min to 0 and any pixel greater than or equal to max to 255.

in[0]:	Input greyscale image.
out[0]:	Output greyscale image.
min:	Any value equal to or less than this is set to 0.
max:	Any value equal to or greater than this is set to 255.

B.8 stereo

This module contains two sets of algorithms, the major set is dense correspondence algorithms, of which there are many variations on the maximum Likelihood Stereo Algorithm[37]. There are also a set of algorithms to render out the resulting correspondence maps, to generate depth maps from the data and to then visualise them. Some noise reduction operations are also provided for depth maps.

stereo.basic.MLM:

```
<op name="stereo.basic.MLM" in="left,right"
out="offsetLeft,offsetRight" sd="2.0" fov="1.5707963267948966"
pd="0.9"/>
```

This impliments the first algorithm given in [37], it takes a left and right greyscale

image (`TimageL`) and outputs a pair of correspondence maps, of type `ToffsetImage`. The default parameters given are the default parameters taken from the paper.

in[0]:	The left greyscale image.
in[1]:	The right greyscale image.
out[0]:	The left output correspondence map.
out[1]:	The right output correspondence map.
sd:	The standard deviation of the Gaussian noise that is considered to be in the image.
fov:	The field of view of the image, in radians. Defaults to 90°
pd:	Percentage of the image that is not occluded, i.e. $1 - pd$ should be the percentage of pixels that are not visible in the other image.

stereo.basic.MLMH:

This is identical to `stereo.basic.MLM` except it implements the second algorithm of the paper

stereo.basic.MLMHV:

This is identical to `stereo.basic.MLM` except it implements the third algorithm of the paper

stereo.colour.MLM:

```
<op name="stereo.colour.MLM" in="left,right,covar"
out="offsetLeft,offsetRight" sd="2.0" fov="1.5707963267948966"
pd="0.9"/>
```

This implements a variation on the basic MLM algorithm by using colour.

in[0]:	The left colour image.
in[1]:	The right colour image.
in[2]:	The covariance matrix that relates the three colour components.
out[0]:	The left output correspondence map.
out[1]:	The right output correspondence map.
sd:	The standard deviation of the Gaussian noise that is considered to be in the image.
fov:	The field of view of the image, in radians. Defaults to 90°
pd:	Percentage of the image that is not occluded, i.e. $1 - pd$ should be the percentage of pixels that are not visible in the other image.

stereo.colour.MLMH:

This is identical to `stereo.colour.MLM` except it implements the second algorithm of the paper

stereo.colour.MLMHV:
This is identical to `stereo.colour.MLM` except it implements the third algorithm of the paper

stereo.area33.MLM:
```
<op name="stereo.area33.MLM" in="left,right,covar"
out="offsetLeft,offsetRight" sd="2.0" fov="1.5707963267948966"
pd="0.9"/>
```

This applies the same expansion that `stereo.colour.MLM` applies, but by increasing it to use a 3x3 window.

in[0]:	The left greyscale image.
in[1]:	The right greyscale image.
in[2]:	The 9x9 covariance matrix that relates the nine pixels inside the window.
out[0]:	The left output correspondence map.
out[1]:	The right output correspondence map.
sd:	The standard deviation of the Gaussian noise that is considered to be in the image.
fov:	The field of view of the image, in radians. Defaults to 90°
pd:	Percentage of the image that is not occluded, i.e. $1 - pd$ should be the percentage of pixels that are not visible in the other image.

stereo.basic.MLM.NCC:
```
<op name="stereo.basic.MLM.NCC" in="left,right"
out="offsetLeft,offsetRight" area="2" sd="2.0"
fov="1.5707963267948966" pd="0.9"/>
```

This achieves the same effect as `stereo.area33.MLM` but instead of extending the MLM algorithm to do area it uses the basic MLM algorithm on values calculated using Normalised Cross Correlation.

in[0]:	The left greyscale image.
in[1]:	The right greyscale image.
out[0]:	The left output correspondence map.
out[1]:	The right output correspondence map.
area:	Defines the window size for the Normalised Cross Correlation, as $(2area + 1)$ square.
sd:	The standard deviation of the Gaussian noise that is considered to be in the image.
fov:	The field of view of the image, in radians. Defaults to 90°

pd: Percentage of the image that is not occluded, i.e. $1 - pd$ should be the percentage of pixels that are not visible in the other image.

stereo.basic.MLMH.NCC:
This is identical to `stereo.basic.MLM.NCC` except it implements the second algorithm of the paper

stereo.basic.MLMHV.NCC:
This is identical to `stereo.basic.MLM.NCC` except it implements the third algorithm of the paper

stereo.depth:

```
<op name="stereo.depth" in="offset1,offset2,proj1,proj2"
out="depth"/>
```

This calculates a depth map from an offset pair and there projection matrices, outputting it as depth for image1.

in[0]: Offset map for camera 1
in[1]: Offset map for camera 2
in[2]: Projection matrix for camera 1
in[3]: Projection matrix camera 2
out[0]: Depth map, `Tdepth`. Note that it has no colour information and needs to be merged with the relevant image if required.

stereo.depth.extrema:

```
<op name="stereo.depth.extrema" in="in" out="out" sd="2.0"/>
```

This calculates the mean and standard deviation of the entire input depth map, it then sets any values outside of `sd` multiplied by the standard deviation from the mean to unknown. This removes extreme values.

in[0]: Input depth map
out[0]: Output depth map
sd: A multiplier to indicate sensitivity.

stereo.depth.norm:

```
<op name="stereo.depth.norm" in="in" out="out" norm="100"
min="-1" max="100"/>
```

This scales a depth map so the given depth is the greatest, it then restricts values to a given range. (This is useful for pushing objects that get too close to the camera away and scaling different sized scenes so they are equally navigable with the visualisation tools.)

in[0]:	Input depth map
out[0]:	Output depth map
norm:	The values are scaled such that this is the maximum depth.
min:	All values less than this value after the scaling are set to this value.
max:	All values greater than this value after the scaling are set to this value.

stereo.depth.outlier:

```
<op name="stereo.depth.outlier" in="in" out="out" size="1"
factor="1.0"/>
```

This attempts to remove outlier points from a given depth map. It calculates the mean and standard deviation of the window with and without the pixel in question. If the gap between the means of this calculation is greater than the standard deviation multiplied by `factor` then it is an outlier and set to unknown. Any point that dose not have at least 3 neighbours in the window is also set to unknown. This algorithm is over zealous, after running it a Gaussian blur should then be run to fill in the holes.

in[0]:	Input depth map
out[0]:	Output depth map
size:	The window size is $(2size + 1)$.
factor:	The scalar factor that decides how sensitive the algorithm is.

stereo.depth.blur:

```
<op name="stereo.depth.blur" in="in" out="out" sd="0.0"
unsd="4.0"/>
```

This applies a Gaussian blue to a depth map. The Gaussian blur only includes values that are set, unknown depth values are ignored. A different standard deviation can be used when the central pixel is known than when it is not. If a standard deviation value is set to 0 it dose not apply that particular one, so by default this calculates unknown depth values as Gaussian weighted averages of there surrounding area but does not affect known values.

in[0]:	Input depth map
out[0]:	Output depth map
sd:	Standard deviation used for the blur when applied to known depth values.
unsd:	Standard deviation used for the blur when applied to unknown depth values.

stereo.chess:

```
<op name="stereo.chess" in="offset" out="image"/>
```

This generates an image from a correspondence map, where it renders a chess board distorted by the correspondence map, as a means of visualising a correspondence map.

in[0]: Offset map
out[0]: Output colour image

stereo.hoffset:

```
<op name="stereo.hoffset" in="offset" out="image" range="24"/>
```

This renders an offset map to an image indicating its horizontal offset. Pixels with no offset are red, they fade towards blue for it heading left and green for it heading right. Unknown correspondences (Occlusions, noise) are rendered black.

in[0]: Offset map
out[0]: Output colour image
range: Maximum offset before it reaches he maximum colour and is clamped.

B.9 3d

This provides two methods, for generating 3D data from depth maps and shaped textures.

3d.from.depth.mesh:

```
<op name="3d.from.depth.mesh" in="depth,proj" out="model"
scale="32"/>
```

This converts a depth map into a textured mesh, constructed from quads.

in[0]: This is the depth map, it should include colour information.
in[1]: A 4x3 projection matrix for the camera, should be calculated from normalised coordinates. ([-1,-1]x[1,1])
out[0]: The output 3D model.
scale: The length of the sides of quads in pixels.

3d.from.shaped.mesh:

```
<op name="3d.from.shaped.mesh" in="shaped" out="model"
scale="32"/>
```

This converts a shaped texture into a textured mesh, constructed from quads.

in[0]:	The shaped texture to use.
out[0]:	The output 3D model.
scale:	The length of the sides of quads in pixels.

B.10 3d.view

This provides visualisation capabilities, as either a point cloud or a textured mesh, using OpenGL.

Both windows use the same interface, it uses the concept of a scene centre, which the camera moves in relation to and always looks at.

- Dragging with the LMB dollies around the scene centre.
- Dragging with the RMB pans the scene, moving the scene centre in the plane perpendicular to the cameras direction.
- The mouse wheel moves in and out from the centre point.
- `Q` and `A` move the scene centre on the X axis.
- `W` and `S` move the scene centre on the Y axis.
- `E` and `D` move the scene centre on the Z axis.
- `G` toggles the rendering of a X-Z grid.
- `O` toggles the rendering of an origin.
- `B` toggles the background colour between black and white. (Black is easier on the eyes, white is suitable for output into this document.)
- `R` resets the camera to the centre of the world looking down the negative Z axis.
- `T` sets the camera to look straight down.
- `C` toggles backface culling on and off.
- `F` toggles if clockwise faces are backfacing or anti-clockwise faces.
- `P` toggles between perspective rendering and orthographic rendering.

3d.view.points:

```
<op name="3d.view.points" in="model"/>
```

Opens a window to display a point cloud.

in[0]: The 3D model to display

3d.view.mesh:

```
<op name="3d.view.mesh" in="model"/>
```

Opens a window to display a textured polygon soup.

in[0]: The 3D model to display

B.11 features

The features library contains corner detectors and corner matchers, to find matches between images that are probably the same point in space.

features.corner.harris:

```
<op name="features.corner.harris" in="image" out="points"
k="0.04" sd="0.7" border="3"/>
```

This implements a harris corner detector[10]. It takes as input a greyscale image and outputs a list of 2D points. In addition to an array of points represented as a `FposX` and `FposY` it outputs them in sorted order, with the strongest point first and the weakest last, so the list can be clipped down to a manageable size sensibly.

in[0]:	The image to detect corners in.
out[0]:	An array of corners.
k:	This is the k value from the algorithm, should probably be left as 0.04.
sd:	Standard deviation of the Gaussian kernel used by the algorithm.
border:	A border in pixels around the image that is ignored, due to the problems borders produce.

features.render.corner:

```
<op name="features.render.corner" in="points,image" out="image"
colour="R" type="P"/>
```

This renders the given corners onto the given image and outputs it, for verification.

in[0]:	The points to be rendered
in[1]:	The image to render onto.
out[0]:	The image with the points rendered onto it.
colour:	This field can contain 'R', 'G' or 'B', as the colour to render the corners as.
type:	This can be either 'P' to render points or 'C' to render crosses.

features.prune.count:

```
<op name="features.prune.count" in="lots" out="less"
count="1000"/>
```

This prunes points by simply clipping the list to the given length.

in[0]:	The point array before being clipped.
out[0]:	The point array after being clipped.
count:	Maximum number of corners to be left after clipping.

features.prune.area:

```
<op name="features.prune.area" in="lots,image" out="less"
divx="32" divy="24" count="4" max="8000"/>
```

This prunes points but instead of having a global count for each square it divides the image up into squares and has a local count for each one, to stop the corners being crowded around the highest contrast features of the image.

in[0]:	The point array before being clipped.
in[1]:	The image, so it knows its dimensions and can divide up accordingly.
out[0]:	The point array after being clipped.
divx:	How many divisions to have on the x axis.
divy:	How many divisions to have on the y axis.
count:	Maximum number of corners to be left in each square after clipping.
max:	Maximum number of corners down the given array to consider, to stop squares in the image that contain no useful content from being filled with corners.

features.match.ncc:

```
<op name="features.match.ncc" in="corner1,image1,corner2,image2"
out="matrix" area="3" max="0.1"/>
```

This produces a similarity matrix between two images corners using Normalised Cross Correlation to compare windows around the corners.

in[0]:	The corners for image 1.

in[1]:	Image 1
in[2]:	The corners for image 2.
in[3]:	Image 2
out[0]:	Similarity matrix.
area:	The NCC is done over a window of $(2area + 1)$.
max:	This defines the maximum distance between two pixels before they are not matched, in terms of a fraction of the image.

features.select.basic:

```
<op name="features.select.basic" in="corner1,corner2,sim_matrix"
out="matrix" cutoff="0.9"/>
```

Given a similarity matrix this produces a data matrix of selected corner pairs that are probably matched. It works by considering any two corners matched if the NCC between them is greater than either corner with any other corner.

in[0]:	The corners for image 1.
in[1]:	The corners for image 2.
in[3]:	The similarity matrix
out[0]:	Data matrix of corner pairs. Each row reads as a match of (image1.x,image1.y,image2.x,image2.y).
cutoff:	Any NCC less than this value is considered to be a mismatch.

features.select.advanced:

```
<op name="features.select.advanced" in="corner1,corner2,sim_matrix"
out="matrix" cutoff="0.9" slice="0.9" set="1.1" iter="3"/>
```

Identical to `features.select.basic` except instead of using the distance cut-off set in `features.match.ncc` it works out an optimum match distance to reduce the number of outliers that get through. It does `iter` iterations, each iteration it gets all the matches within the current distance (Starts as the width of the image.) and generates a histogram of distances between them. It then takes the first `slice` fraction of the inliers and sets the distance to `set` multiplied by the largest distance in this set. The matches selected in the final iteration are output.

in[0]:	The corners for image 1.
in[1]:	The corners for image 2.
in[3]:	The similarity matrix
out[0]:	Data matrix of corner pairs. Each row reads as a match of (image1.x,image1.y,image2.x,image2.y).
cutoff:	Any NCC less than this value is considered to be a mismatch.
slice:	Fraction of matches considered to probably be inliers.
set:	A multiplier used by the algorithm.
iter:	How many iterations of refinement to do.

features.render.match:

```
<op name="features.render.match" in="data_matrix,image1,image2"
out="image"/>
```

This renders matches between two images in such a way they can be checked. It renders image1 to the red channel, image2 to the blue channel and then draws green lines between the matches.

in[0]:	Data matrix of the matches.
in[1]:	Image 1, greyscale.
in[2]:	Image 2, greyscale.
out[0]:	The rendered image.

B.12 camera

The camera module is concerned with calculating the fundamental matrix and rectification. It also provides lots of validation operations.

camera.ext.match.fun:

```
<op name="camera.ext.match.fun" in="match,image1,image2"
out="fun[,valid]" cutoff="1.0" cert="0.95" max="1000000"
refine="2"/>
```

It takes a data matrix of matches and outputs the fundamental matrix. It uses the RANSAC solution of trying a random set of 8 matches and generating a fundamental matrix from it. It then uses the epipolar constraint to define which points are inliers and which are outliers. This is done a number of times and the points defined as inliers for the largest set are ultimately then chosen as the correct point set. The fundamental matrix is then calculated for all of these inliers to produce the final fundamental matrix which is output.

in[0]:	Data matrix of the matches.
in[1]:	Image 1, simply for its dimensions so the matches can be normalised.
in[2]:	Image 2, again for its dimensions.
out[0]:	The 3x3 fundamental matrix.
out[1]:	(Optional) The data matrix passed in but only with the corners defined as inliers by this process.
cutoff:	The distance a point can be from the epipolar line in pixels for it to be considered as on the epipolar line and therefore a match.
cert:	The chance of having the correct solution, it keeps running till it is this sure it has the correct answer.

max: The max number of runs through, in case its chances of success simply aren't increasing fast enough.
refine: The number of refinement stages to have after the RANSAC process is over.

camera.rectify.polar:

```
<op name="camera.rectify.polar" in="image1,image2,funmat"
out="rect1,rect2"/>
```

This implements polar rectification[26]. Saves the information required to de-rectify in the output so the de-rectification algorithm is simple.
in[0]: Image 1.
in[1]: Image 2.
in[2]: 3x3 fundamental matrix.
out[0]: Rectified version of image 1.
out[1]: Rectified version of image 2.

camera.derectify:

```
<op name="camera.derectify" in="image1,rect1,rect2,corr1"
out="corr_out"/>
```

This de-rectifies rectified correspondence data.
in[0]: Image 1.
in[1]: Rectified image 1.
in[2]: Rectified image 2.
in[3]: Rectified Correspondences for image 1
out[0]: De-rectified version of correspondences for image 1.

camera.render.epilines:

```
<op name="camera.render.epilines" in="image,funmat,match"
out="image" transpose="false"/>
```

This takes the given image and renders the epipolar lines associated with the points in match onto it using the given fundamental matrix, for validation.
in[0]: Image.
in[1]: 3x3 fundamental matrix.
in[2]: Match array.
out[0]: Image with epipolar lines rendered onto it.
transpose: If the fundamental matrix goes from this image to another then set transpose to false, if its from another to this set trasnpose to true.

camera.projection.gen:

```
<op name="camera.projection.gen" in="int1,fun,int2"
out="proj1,proj2"/>
```

Given the intrinsic matrices for two cameras and the fundamental matrix between two views this generates the projection matrices of teh two views assuming that the first view is at the origin.

in[0]: Intrinsic matrix for the first view.
in[1]: 3x3 fundamental matrix.
in[2]: Intrinsic matrix for the second view.
out[0]: Projection matrix for the first view.
out[1]: Projection matrix for the second view.

camera.match.cloud:

```
<op name="camera.match.cloud" in="proj1,proj2,image1,image2,matches"
out="model"/>
```

Given the projection matrices and images of two views it then converts a set of matches between the two views into a point cloud. Used for testing purposes.

in[0]: Projection matrix for the first view.
in[1]: Projection matrix for the second view.
in[2]: Image 1.
in[3]: Image 2.
in[4]: Data matrix of matches.
out[0]: 3D model of vertices.

B.13 intrinsic

The intrinsic module provides tools to obtain a cameras intrinsic parameters.

intrinsic.select.squares:

```
<op name="intrinsic.select.squares" in="image,corners"
out="squares"/>
```

This is given an image and corners on it. It provides an interface by which the user can select the corners of a calibration grid, working through the grid in reading order, it snaps selected corners to the given list of detected corners, so the user does not have to be accurate. The interface is LMB to place next point, RMB jumps closest point to where you click to where you click, to correct mistakes. Note that the moment you place the last point the interface exits. The corners are colour coded such that you will know if you miss a corner/double click.

in[0]: The image to select corners for.

in[1]: The computer detected corners.
out[0]: List of corners that make up the calibration grid.

intrinsic.select.squares:

```
<op name="intrinsic.select.squares" in="image,corners"
out="image"/>
```

This renders the result of `intrinsic.select.squares` to an image for verification.

in[0]: The image to render to.
in[1]: The corners to render.
out[0]: The output image with the calibration grid on.

intrinsic.homography.squares:

```
<op name="intrinsic.homography.squares" in="image,squares"
out="matrix,[render]"/>
```

This calculates the homography required to get from the 3D coordinates of the points on the square calibration grid to the image points. Essentially it works out the projection matrix based on us having a known set of points. The passed in image is just to normalise the image coordinates.

in[0]: The image, just for normalising the coordinates.
in[1]: The corners that make up the calibration grid.
out[0]: 3x3 homography matrix.
out[1]: (Optional.) If given it renders the matched calibration target to the given image, for seeing what the error is.

intrinsic.calc:

```
<op name="intrinsic.calc" in="homographys" out="intrinsic"/>
```

Given a set of homographies this calculates an intrinsic matrix, at least three must be provided. (Use `var.append` to construct the input.)

in[0]: An array of homographys.
out[0]: A 3x3 intrinsic matrix.

B.14 registration

This section was ultimately meant to contain registration algorithms, but due to time constraints instead it contains a set of algorithms to manage shaped tex-

tures.

registration.correspondence.to.3D:

```
<op name="registration.correspondence.to.3D"
in="image1,offset1,offset2,proj1,proj2" out="shaped"/>
```

This generates a shaped texture for an image given correspondence maps and other details.

in[0]:	Image to texture the shaped texture.
in[1]:	Offset map from view 1 to view 2.
in[2]:	Offset map from view 2 to view 1.
in[3]:	Projection matrix for view 1.
in[4]:	Projection matrix for view 2.
out[0]:	A shaped texture.

registration.clean.border:

```
<op name="registration.clean.border" in="in" out="out"
border="10"/>
```

This removes a border of pixels around the edge of a shaped texture as there usually wrong.

in[0]:	Input shaped texture.
out[0]:	Output shaped texture.
border:	Border to cull.

registration.clean.average:

```
<op name="registration.clean.average" in="in,proj" out="out"
average="40.0"/>
```

This scales a shaped texture such that the average distance from the camera is as given, for visualisation.

in[0]:	Input shaped texture.
in[1]:	Projection matrix, so it knows where the centre of the camera is.
out[0]:	Output shaped texture.
average:	Average distance for parts of the shaped texture to be from the camera.

registration.clean.merge:

```
<op name="registration.clean.merge" in="in" out="out" area="4"
mult="2.0" cutoff="50"/>
```

This removes noise from shaped textures, and if given several merges them to

assist with noise reduction. For each shaped texture it takes an average and standard deviation of a window, and points outside `mult` times the standard deviation are declared as outliers. If there are at least cut-off inliers the value is then set to the average of the windows excluding the outliers. It then merges shaped textures using a kalman filter.

in[0]:	Input shaped texture.
in[1]:	Projection matrix, so it knows where the centre of the camera is.
out[0]:	Output shaped texture.
average:	Average distance for parts of the shaped texture to be from the camera.

C Testing

C.1 XML Parsing

C.1.1 Correct files

ok1.xml:

```
<a/>
```

ok2.xml:

```
<elem attrib="a"/>
```

ok3.xml:

```
<elem attrib="12345">
</elem>
```

ok4.xml:

```
<elem>
    <!- A coment ->
    Some text
</elem>
```

ok5.xml:

```
<elem
attrib="12345">
    Some text
    <turnip b="9876"/>
    Some more text
    <turnip>
        Text
        <cabbage size="5.6"/>
    </turnip>
</elem>
```

C.1.2 Incorrect files

bad1.xml:

bad3.xml:
```
Outside
<elem/>
```

bad5.xml:
```
elem/>
```

bad7.xml:
```
<elem attrib="/>
```

bad9.xml:
```
<rootone/>
<roottwo/>
```

bad2.xml:
```
<elem>
```

bad4.xml:
```
<elem/
```

bad6.xml:
```
<elem attrib/>
```

bad8.xml:
```
<elem>
hello<world
</elem>
```

bad10.xml:
```
<elem>
     <!- Not closed
     <var attrib="mouse"/>
</elem>
```

C.2 Framework

script1.xml:
```
<op>
  <op name="msg" msg="Hello World!"/>
</op>
```

script2.xml:
```
<op>
  <var name="image"/>

  <op name="image.io.load" out="image">
    <file value="test.jpg"/>
  </op>

  <op name="image.io.save" in="image">
    <file value="out1.jpg"/>
  </op> </op>
```

script3.xml:

```
<op>
  <var name="image"/>

  <op uses="image">
    <var name="local"/>

    <op name="image.io.load" out="local">
      <file value="test.jpg"/>
    </op>

    <op name="var.copy" in="local" out="image"/>
  </op>

  <op name="image.io.save" in="image">
    <file value="out2.jpg"/>
  </op>
</op>
```

script4.1.xml:

```
<op>
  <var name="image"/>

  <op name="image.io.load" out="image">
    <file value="test.jpg"/>
  </op>

  <op name="var.save" in="image" filename="store.var"/>
</op>
```

script4.2.xml:

```
<op>
  <var name="image"/>

  <op name="var.load" out="image" filename="store.var"/>

  <op name="image.io.save" in="image">
    <file value="out3.jpg"/>
  </op>
</op>
```

C.3 Increment 1

C.3.1 POV Ray Scripts

This section gives the POV ray source used to render the CG test scenes. (Using POV Ray's radiosity renderer, at 800x600.) They all start with the following:

```
#include "colors.inc"
#include "textures.inc"
#include "rad_def.inc"

global_settings
{
 radiosity
 {
  Rad_Settings(Radiosity_Final,false,false)
 }
}
```

Each scene gives two camera settings, first for the left then second for the right.
I1, 'Planes':

```
camera
{
 location <0.0, 10, 0>
 look_at  <0.0, 0, 0>
}

camera
{
 location <0.5, 10, 0>
 look_at  <0.5, 0, 0>
}

plane
{
 <0, 1, 0>, 0
 texture {Cork}
 finish {ambient 0.0}
}

box
{
 <-3, 1,-2>,<3,1.1,2>
```

```
 texture {Cork}
 finish {ambient 0.0}
}

sky_sphere
{
 pigment {colour White}
}
```

I2, 'Curves':

```
camera
{
 location <0.0, 10, 0>
 look_at  <0.0, 0, 0>
}

camera
{
 location <0.5, 10, 0>
 look_at  <0.5, 0, 0>
}

sphere
{
 <0, 0, 0>, 2
 texture
 {
  Rusty_Iron
  finish {phong 1 ambient 0.0}
 }
}

torus
{
 3.5, 1
 pigment {DMFDarkOak}
 finish {ambient 0.0}
}

#declare i = 0;
#while (i<4)
torus
{
 0.7, 0.4
```

```
 rotate x*90
 translate <2,0,0>
 rotate y*90*i
 texture {DMFWood6}
 finish {ambient 0.0}
}
#declare i = i+1;
#end

sky_sphere
{
 pigment {colour White}
}
```

I3, 'Objects':

```
camera
{
 location <-0.25, 6, 5>
 look_at  <-0.25, 0, 0>
}

camera
{
 location <0.0, 6, 5>
 look_at  <0.0, 0, 0>
}

plane
{
 <0, 1, 0>, 0
 texture {DMFLightOak}
 finish {ambient 0.0}
}

#declare i = 0;
#while (i<8)
box
{
 <-0.5,0,-0.5>,<0.5,1,0.5>
 texture {PinkAlabaster}
 finish {ambient 0.0}
 translate <2,0,0>
 rotate y*45*i
}
```

```
#declare i = i+1;
#end

#declare i = 0;
#while (i<12)
box
{
 <-0.5,0,-0.5>,<0.5,2,0.5>
 texture {DMFLightOak}
 finish {ambient 0.0}
 translate <3.5,0,0>
 rotate y*30*i
}
#declare i = i+1;
#end

#declare i = 0;
#while (i<4)
lathe
{
 cubic_spline
 8,
 <0.1,0>, <0.1,0>, <0.1,0.4>, <0.15,0.45>,
 <0.4,0.6>, <0.15,0.75>, <0.0,0.8> <0.0,0.8>
 texture {Cork}
 finish {ambient 0.0}
 translate <2,1,0>
 rotate y*90*i
}
sphere
{
 <2, 1.5, 0>, 0.2
 texture {Gold_Metal}
 finish {ambient 16}
 rotate y*90*i
}
#declare i = i+1;
#end

torus
{
 20, 1
 pigment {rgb <57/255,126/255,175/255>}
 finish {ambient 100.0}
```

```
 translate <0,1,0>
}

sphere
{
 <0, 0, 0>, 1
 texture {Cork}
 finish {ambient 0.0}
}

sky_sphere
{
 pigment {colour White}
}
```

C.4 Increment 2

C.4.1 Rectification Script

The following code uses PHP[1], a scripting language that usually runs on web servers to generate http pages, but is equally good at anything where text output is the goal.

```
<?
 echo("<op>\n");

 $imageCount = 21;

 for ($i=1;$i<=$imageCount;$i++)
 {
  echo("\t<var name=\"i$i\"/>\n");
  echo("\t<op name=\"image.io.load\" out=\"i$i\">
        <file value=\"$i.jpg\"/></op>\n");
  echo("\t<var name=\"i$i"."g\"/>\n");
  echo("\t<op name=\"image.filter.grey\" in=\"i$i\"
        out=\"i$i"."g\"/>\n");
  echo("\t<var name=\"i$i"."cm\"/>\n");
  echo("\t<op name=\"features.corner.harris\" in=\"i$i"."g\"
        out=\"i$i"."cm\" sd=\"0.7\" border=\"3\" res=\"1\"/>\n");
  echo("\t<var name=\"i$i"."c\"/>\n");
  echo("\t<op name=\"features.prune.area\" in=\"i$i"."cm,i$i"."g\"
        out=\"i$i"."c\" divx=\"32\" divy=\"24\" count=\"4\"
```

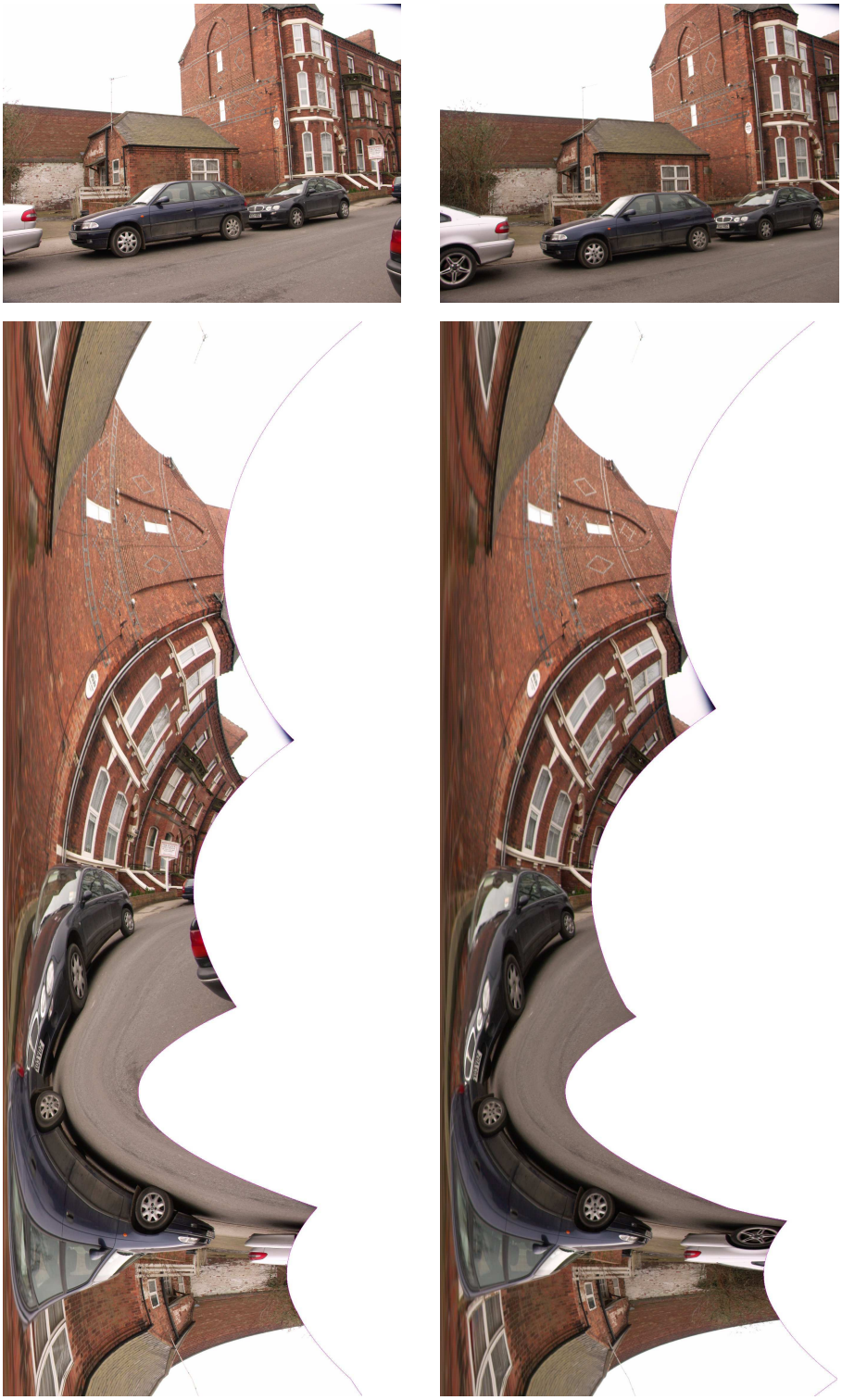
[1] http://www.php.net/

```
        max=\"8000\"/>\n");
  echo("\n");
 }

 for ($i=1;$i<=$imageCount;$i++)
 {
  for ($j=$i+1;$j<=$imageCount;$j++)
  {
   echo("\t\t<op name=\"msg\" msg=\"$i-$j...\"/>\n");
   echo("\t<op uses=\"i$i,i$i"."c,i$i"."g,i$j,i$j"."c,i$j"."g\">\n");
   echo("\t\t<var name=\"ncc\"/>\n");
   echo("\t\t<op name=\"features.match.ncc\"
         in=\"i$i"."c,i$i"."g,i$j"."c,i$j"."g\" out=\"ncc\"
         area=\"3\" res=\"1\" max=\"0.1\"/>\n");
   echo("\t\t<var name=\"mat\"/>\n");
   echo("\t\t<op name=\"features.select.advanced\"
         in=\"i$i"."c,i$j"."c,ncc\" out=\"mat\" cutoff=\"0.9\"
         slice=\"0.9\" set=\"1.1\" iter=\"10\"/>\n");
   echo("\t\t<var name=\"fun\"/>\n");
   echo("\t\t<op name=\"camera.ext.match.fun\" in=\"mat,i$i,i$j\"
         out=\"fun\" cutoff=\"0.5\" cert=\"0.95\" max=\"10000\"
         refine=\"2\"/>\n");
   echo("\t\t<var name=\"rect-a\"/>\n");
   echo("\t\t<var name=\"rect-b\"/>\n");
   echo("\t\t<op name=\"camera.rectify.polar\" in=\"i$i,i$j,fun\"
         out=\"rect-a,rect-b\"/>\n");
   echo("\t\t<op name=\"image.io.save\" in=\"rect-a\">
         <file value=\"rect$i-$j-a.jpg\"/></op>\n");
   echo("\t\t<op name=\"image.io.save\" in=\"rect-b\">
         <file value=\"rect$i-$j-b.jpg\"/></op>\n");
   echo("\t</op>\n");
  }
 }

 echo("</op>\n");
?>
```

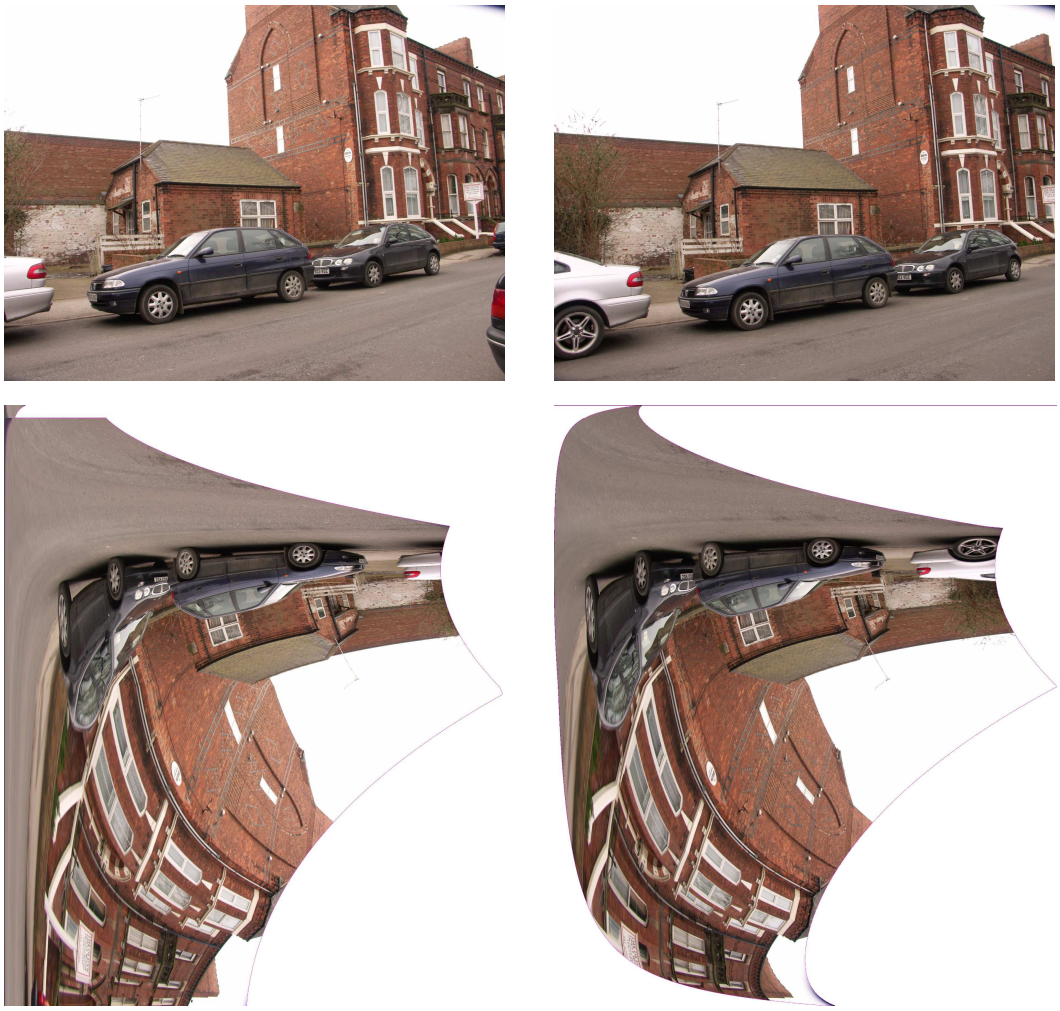
C.4.2 Rectification Special Cases

Each of the below tests is arranged with the source pair at the top then the rectified pair below them. The top left image is the same in all cases.

Rectification where both epipoles are in the image:



Rectification where one epipole is in the image and the other is outside the image:



Rectification where both epipoles are outside the image:

C.5 Matching Limits Test

C.5.1 Script

The following PHP code was used to generate the testing script.

```
<?
 echo("<op>\n");

 $imageCount = 45;

 echo("\t<var name=\"base\"/>\n");
```

```
echo("\t<op name=\"image.io.load\" out=\"base\">
      <file value=\"0.jpg\"/></op>\n");
echo("\t<var name=\"base-grey\"/>\n");
echo("\t<op name=\"image.filter.grey\" in=\"base\"
      out=\"base-grey\"/>\n");
echo("\t<var name=\"base-cm\"/>\n");
echo("\t<op name=\"features.corner.harris\" in=\"base-grey\"
      out=\"base-cm\" sd=\"0.7\" border=\"3\" res=\"1\"/>\n");
echo("\t<var name=\"base-c\"/>\n");
echo("\t<op name=\"features.prune.area\" in=\"base-cm,base-grey\"
      out=\"base-c\" divx=\"32\" divy=\"24\" count=\"4\"
      max=\"8000\"/>\n");
echo("\n");

for ($i=1;$i<=$imageCount;$i++)
{
 echo("\t<op uses=\"base-grey,base-c\">\n");

 echo("\t\t<var name=\"img\"/>\n");
 echo("\t\t<op name=\"image.io.load\" out=\"img\">
       <file value=\"$i.jpg\"/></op>\n");
 echo("\t\t<var name=\"img-grey\"/>\n");
 echo("\t\t<op name=\"image.filter.grey\" in=\"img\"
       out=\"img-grey\"/>\n");
 echo("\t\t<var name=\"img-cm\"/>\n");
echo("\t\t<op name=\"features.corner.harris\" in=\"img-grey\"
      out=\"img-cm\" sd=\"0.7\" border=\"3\" res=\"1\"/>\n");
 echo("\t\t<var name=\"img-c\"/>\n");
 echo("\t\t<op name=\"features.prune.area\" in=\"img-cm,img-grey\"
       out=\"img-c\" divx=\"32\" divy=\"24\" count=\"4\"
       max=\"8000\"/>\n");
 echo("\n");

 echo("\t\t<op name=\"msg\" msg=\"$i...\"/>\n");
 echo("\n");

 echo("\t\t<var name=\"ncc\"/>\n");
 echo("\t\t<op name=\"features.match.ncc\"
       in=\"base-c,base-grey,img-c,img-grey\" out=\"ncc\"
       area=\"3\" res=\"1\" max=\"0.1\"/>\n");
 echo("\t\t<var name=\"mat\"/>\n");
 echo("\t\t<op name=\"features.select.advanced\" in=\"base-c,img-c,ncc\"
       out=\"mat\" cutoff=\"0.9\" slice=\"0.9\" set=\"1.1\" iter=\"10\"/>\n");
 echo("\t\t<var name=\"fun\"/>\n");
```

```
  echo("\t\t<var name=\"inliers\"/>\n");
  echo("\t\t<op name=\"camera.ext.match.fun\" in=\"mat,base-grey,img-grey\"
        out=\"fun,inliers\" cutoff=\"0.5\" cert=\"0.95\" max=\"100000\"
        refine=\"2\"/>\n");
  echo("\n");

  echo("\t\t<var name=\"inliers-rend\"/>\n");
  echo("\t\t<op name=\"features.render.match\"
        in=\"inliers,base-grey,img-grey\" out=\"inliers-rend\"/>\n");
  echo("\t\t<op name=\"image.io.save\" in=\"inliers-rend\">
        <file value=\"$i-match.jpg\"/></op>\n");

  echo("\t</op>\n\n");
 }

 echo("</op>\n");
?>
```

C.5.2 Images

Of the following two images the left image is the 0° image and the right image is the 4° matched image. (The matched image has the 0° image in the red channel, the 4° image in the blue channel and green lines connecting matches.)